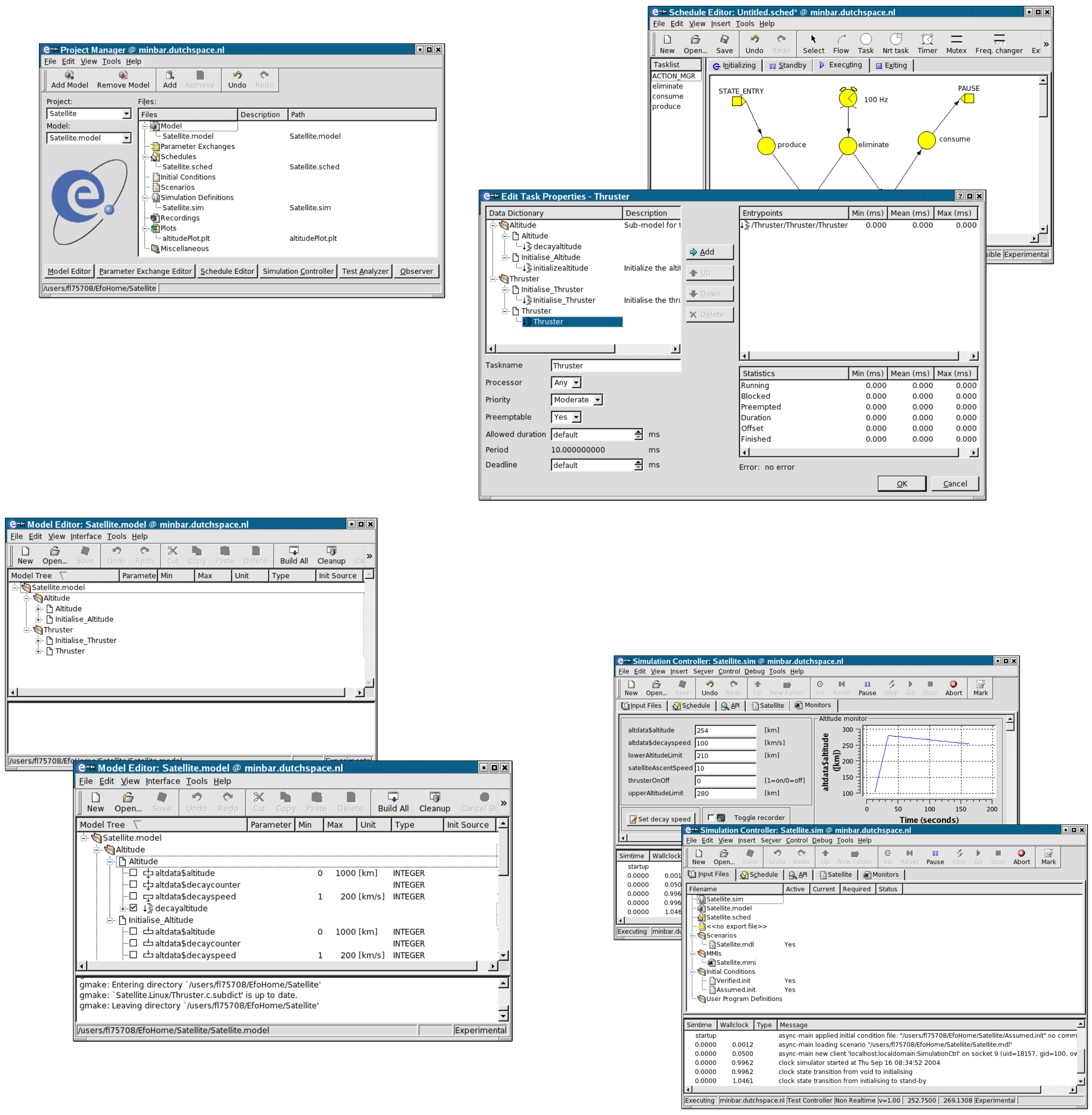
EuroSim Mk4.0
Software User's Manual



Project Manager @ minbar.dutchspace.nl
Schedule Editor: Untitled.sched* @ minbar.dutchspace.nl
Edit Task Properties - Thruster
Model Editor: Satellite.model @ minbar.dutchspace.nl
Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
Altitude monitor

```
gmake: Entering directory `/users/fl75708/EfoHome/Satellite'
gmake: `Satellite.Linux/Thruster.c.subdict' is up to date.
gmake: Leaving directory `/users/fl75708/EfoHome/Satellite'
```

```
Simtime   Wallclock  Type  Message
 startup                   async-main applied initial condition file: "/users/fl75708/EfoHome/Satellite/Assumed.init" no comm
 0.0000    0.0012          async-main loading scenario "/users/fl75708/EfoHome/Satellite/Satellite.mdl"
 0.0000    0.0500          async-main new client 'localhost.localdomain:SimulationCtrl' on socket 9 (uid=18157, gid=100, ow
 0.0000    0.9962          clock simulator started at Thu Sep 16 08:34:52 2004
 0.0000    0.9962          clock state transition from void to initialising
 0.0000    1.0461          clock state transition from initialising to stand-by
```

National Aerospace Laboratory NLR

Dutch Space

Atos Origin

Summary

EuroSim Mk4.0 is an engineering simulator to support the design, development and verification of space (sub) systems defined by ESA programmes of various scales. The facility provides a reconfigurable real-time execution environment with the possibility of man-in-the-loop and/or hardware-in-the-loop additions.
This document describes the facilities available for usage in EuroSim Mk4.0, and how those facilities can be used.

Revision Record

Issue	Revision	Date	Reason for change	Changes
0	1	11-Mar-1994	Document creation; internal distribution only.	All pages.
0	2	10-Apr-1994	Update to expand contents and take into account internal comments.	
0	3	15-Dec-1994	Completely updated for Mk0.1.	All pages.
0	4	7-Feb-1995	Continued updating of issue 0 revision 3.	All pages.
0	5	25-Apr-1995	Issued for DD/R EuroSim Mk0.1. SR/R-1-RID-SRD-74, SR/R-2-RID-SRD-5, AD/R-2-RID-Model_ICD-2, AD/R-2-RID-Model_ICD-3, AD/R-2-RID-SRD-3, AD/R-2-RID-SUM-2.	All pages.
1	0	18-May-1995	Completely revised to take into account internal comments.	All pages.
1	1	26-Jun-1995	Updated after DDR.	
2	0	15-Jul-1996	New document for EuroSim Mk0.2, reference number of document changed to NLR-EFO-SUM-2.	All pages.
2	1	16-Dec-1996	Issued for DD/R EuroSim Mk0.2. Internal review comments processed. SPRs implemented: 166, 364, 370, 380, 397, 406, 462, 475, 484, 571, 574, 578, 603, 612, 629, 633, 652, 657, 712, 814, 840, 960, 961, 1010, 1011, 1045, 1205, 1216, 1273, 1293, 1326, 1483.	
2	2	17-Feb-1997	Updated after DD/R; the following RIDs have been implemented: 53..67, 69..76, 78..102, 104, 106..123, 125..164, 166..187, 202..209, 211..214, 216..224, 226..251, 255..257, 260, 263, 266..269. Note that not-implemented RIDs from the 200 range have been re-issued for the delta DD/R.	
2	3	25-Apr-1997	Updated after delta DD/R; the following RIDs have been implemented: 42..44, 47..51, 53..68, 72..83, 85..88, 90, 91, 93..102, 104..106, 107 (partly), 108..116, 119..123	
2	4	1-May-1997	EuroSim Mk1 SUM. Inclusion of IGS information/references: reference to IGS SUM, inclusion of IGS overview, definition of IGS interfaces within EuroSim (action IGS-PM7-3). Approved RIDs from DD/R: 68, 77, 103, 105, 124, 125.	

Issue	Revision	Date	Reason for change	Changes
2	5	24-Jun-1997	Added RID numbers for revisions 2 and 3 above. Approved SPRs implemented: 1557, 1549, 1592. Update Test Analyzer section in accordance with SPR-1505, 1651. Updated appendix on MDL following DD/R RID 177 and DD/R RID 103. Also some knock-on changes in Mission Tool Reference.	
3	0	2-Mar-2000	Mk2 release. SPR 1633.	Section 3.2.
3	1	2-May-2000	Mk2rev1 release: Event counter functions added to EuroSim Services. High resolution and max number processors changes added. Recorder file switching and Stimuli cycling changes documented. *HLA extension: EsimRTI* usage as appendix added.	
3	2	6-Oct-2000	Mk2rev2 release: Add appendix describing the run-time interface as used by the test controller. Add appendix explaining AFAP scheduling pitfalls.	
4	0	14-May-2002	Mk3 release: Updated the manual to conform to the new Graphical User Interface.	All pages.
4	1	12-Sep-2003	Mk3rev1 release: Converted to LaTeX. Updated screenshots. Update descriptions of publish functions (API headers). Added description on new 'diff with' functionality (GUI). Added action button support (Simulation Controller). Added description for timebar (Schedule Editor). Added section on user defined EuroSim compatible devices (HW). Updated MDL syntax description. Added chapter for Windows COM. interface.	All pages.
4	2	2-Sep-2004	Mk3rev2 release: Added new chapters for Model Description Editor and Parameter Exchange Editor. Simulation Controller: added description for exports file, removed sections on IGS. Schedule Editor: added description on how to add Parameter Exchange file(s) to the schedule. Model Editor: Added the Model Description file node. EuroSim files and formats: Added Model Description and Parameter Exchange files. Updated screen shots.	All pages.
5	0	18-Apr-2006	Mk4rev0 release: Added new chapters for Calibration Editor, SMP2 Editor and the Web Interface. Model Editor: Added the SMP2 Catalogue file node. Updated various screen shots.	Various pages.

Table of Contents

Part I

EuroSim Basics

Chapter 1

Introduction

1.1 Purpose

The purpose of this document is to provide a user of the EuroSim facility with an understanding of the functions available and the logical order in which they should be used in order to achieve the objective of developing and executing a simulation model for a particular application.
It is expected that the user has some basic UNIX knowledge and familiarity with simulation in general.
This manual is also available on-line, including hypertext.

1.2 Scope

This document describes the use of the EuroSim facility. It provides details of the functions that are available for the user, and relates these functions to a typical operational scenario. It also provides guidance on the development of the application model itself, including the recommended structure of the model, and the library routines provided by the facility.
In this manual the main functions of the EuroSim facility are described from the user's point of view. The document is divided in four parts: the first part, Chapter 2 and Chapter 3, describes the general functionality of EuroSim, its user interface and some of the underlying concepts.
Next, Chapter 4, contains a complete case study for building a working simulator. The more basic functions are described here, but not in detail.
For more detail, chapters 5 through 13, contain a reference guide to all menu items, concepts and objects which can be found in the various editors and windows of EuroSim.
Chapters 16 through 18 contain information on using hardware in the loop with EuroSim.
Finally, a number of appendices contain the remaining information. Abbreviations and terms are defined in Appendix A and Appendix B respectively. The remaining appendices go into more detail on some of the features of EuroSim.

1.3 Where to start

Novice users should start with Chapter 2, and then follow (and possibly re-create) the case study from Chapter 4. It might be necessary to read Chapter 3 to get acquainted with some of EuroSim's user interface aspects.
Users who already have knowledge of EuroSim can immediately proceed to the reference chapters, where each of the EuroSim tools is described in detail.
The table of contents and the index can be used to find certain subjects in the user manual.
Facility managers are advised to read also [OM05], the *EuroSim Owner's Manual*. More files and documents that contain information related to EuroSim can be found in the bibliography.

1.4 Document conventions

The selection of a menu option from the GUI is referred to as for example 'Select the menu option *File:Close*', which means to select from the menu with the name *File* the option *Close*.
Key combinations are shown as 'Alt+Backspace', which means to hold down the key labeled Alt and then simultaneously pressing the Backspace key.
Computer input and output is shown as a `fixed pitch` font. Buttons are referenced with their **label** in bold face.

Chapter 2

Concepts

This chapter introduces the concepts and elements which are common to EuroSim. These include version management and the API interface. Concepts and elements specific to an EuroSim tool or editor are described in the reference chapters for these tools and editors.

2.1 EuroSim simulation life-cycle

The EuroSim simulation life-cycle is executed when going from model code to test analysis.

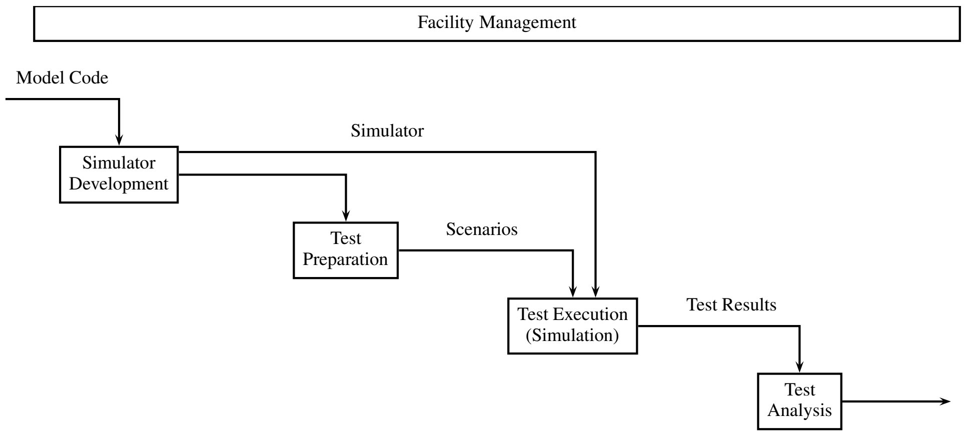


Figure 2.1: EuroSim simulation life cycle

The life-cycle is summarized in Figure 2.1 (no feedback loops are displayed). In the figure, the following phases are shown:

Simulator development
: In this phase the model is assembled from existing submodels, or build from scratch. Existing simulator code can be integrated into the model. Also, an executable version of the simulator is created, including the scheduling of simulator tasks.

Test preparation
: During this phase scenarios for a particular simulator are defined, including stimuli, initial conditions, recording and on-line monitoring requirements.

Test execution
: During this phase the simulator is being run, and the execution of the simulator is monitored.

Test analysis
: During this phase the results from the simulator run are processed and analyzed.

The last rectangle, facility management, offers services with respect to project management and management of EuroSim software and hardware configurations. For more information on facility management, refer to [OM05].
For each of these phases, one or more tools are available to the user. See Section 2.3 for more details.

2.2 Simulator elements

During this life-cycle, a number of objects are used to represent various parts of the simulation. These are:

- A *model*.
- One or more *tasks* and a *schedule*.
- A *data dictionary*.
- One or more *simulation definitions*.
- The *simulator* itself.

Each of these objects is described in more detail in the following sections.

2.2.1 The model

The model (or 'application model') contains all the information needed to describe a real-world system for the purpose of simulation. Using a hierarchical structure, this information comprises of (sub)system descriptions (using any of the languages supported by EuroSim: C, Fortran and Ada-95[1], timing information (through tasks and a schedule) and information on parameters and variables which can be modified or monitored during a simulation (the data dictionary).
The model hierarchy can be used to group common elements together. To this end, the model hierarchy is a tree-like structure (with the model itself at the top), with the various (sub)system descriptions grouped together by nodes in the tree.
The model hierarchy itself is created with the Model Editor (see Chapter 6).

2.2.2 Tasks and schedule

The timing information of a model is defined through one or more tasks and a schedule which ties together the tasks. A task is a sequential list of operations provided by the (sub)systems of the model. These operations have to be executed consecutively, starting with the first operation, and ending with the last one. Within a task, there are no timing constraints and/or synchronization points.
The schedule contains information on when and how tasks should be activated in order to:

- achieve real-time, parallel, simulation when executing the simulation, and
- realize a requested change in simulator state (e.g. from executing to standby); see Section 2.2.5 for more information on simulator states.

The tasks and schedule are defined using the Schedule Editor (see Chapter 11), which is available through the Project Manager.

[1]Ada-95 is not supported in the Windows NT version.

2.2.3 The data dictionary

During a simulation, data can be monitored and/or recorded, and parameters can be set. The data elements which should be accessible during the simulation have to be defined in the data dictionary for this purpose. This is done through the use of so-called API headers (see also Section 2.5).
The data dictionary is defined using the Model Editor (see Chapter 6). Browsing the data dictionary can be done using the Dictionary Browser (see Section 12.5) which is available in several of the editors and tools.

2.2.4 Simulation definition

A simulation definition contains all information required during a simulation: this can be any number of monitors (for monitoring variables), recorders (for storing variable values in a file), stimuli (to simulate external inputs to the simulation), scripts and events (to manually influence the simulation) and initial conditions (to initialize the simulation in a certain state).
More than one simulation definition can be defined for a particular model, each resulting in a different simulation result.
Simulation definitions are created using the Simulation Controller, which is described in Chapter 12.

2.2.5 The simulator

A simulator is one or both of a hardware device and a computer program built out of model-dependent software (i.e. the model code itself, the schedule and the data dictionary) and the model-independent software for the performance and control of the simulation (i.e. the EuroSim provided software). A simulator together with a simulation definition can be used to start a simulation run.
The simulator is always in one of 5 predefined states (see Figure 2.2). These states determine the current phase in the general process of simulation. These same states (except the unconfigured state) are also used within the Schedule Editor to define the schedule.

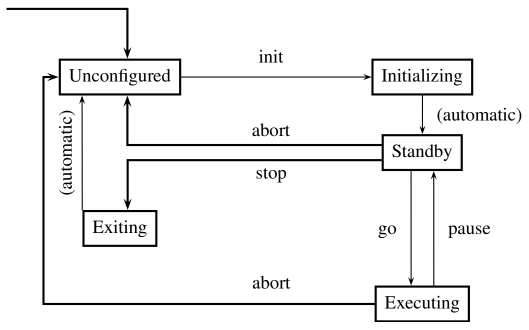


Figure 2.2: Simulator states

State transitions can be triggered by issuing a state transition command, either from the Simulation Controller, the model, or the schedule. The labels in Figure 2.2 correspond to the buttons available in the Simulation Controller (see Section 12.3.1) as well as the MDL commands (see Appendix E). The only missing state transition is the reset as it is too complicated to put in the drawing. Reset can be issued from standby state and is a combination of a stop and an init command where the simulation is not completely stopped and restarted.
The simulator can be run in one of two modes: *real time* or *non-real time*. When a simulation is started in non-real time, the simulation server will try to run the simulation as close to real time as possible. This means that task timing overruns in the simulation will not generate real-time errors. Also, a simulation running non-real time will not claim a whole simulation server: other simulations can also be running (also non-real time). In non-real time mode, it is also possible to instruct EuroSim to run the simulation as fast as possible (see Section 12.7.4 for more information).

2.3 Services and tools

EuroSim offers users two levels of support:

- The first level of support is through a number of tools which can be used to define the simulation. These tools all have an (often graphical) user interface and include editors such as the Model Editor and the Schedule Editor.
- The second level of support is through a number of services which are available to the model developer. Services are functions in the EuroSim software that can be called from within model code. See Section 2.5 and Appendix D.

In the next sections, an overview is given of the available tools.

2.3.1 Project Manager

The Project Manager is used to define new projects. The Project Manager is the main EuroSim window, and is described in detail in Chapter 5.

2.3.2 Model Editor

The Model Editor is used to define a model and its hierarchy together with the definition of the variables and parameters that are available for monitoring, recording, etc. during the simulation run.
The Model Editor is described in detail in Chapter 6.

2.3.3 Model Description Editor

The Model Description Editor is used when integrating several independent models into one simulator without wanting to do the integration explicitly in (model) source code. It is used to describe which model variables should appear in the so called “datapool”.
The Model Description Editor is described in detail in Chapter 7.

2.3.4 Parameter Exchange Editor

The Parameter Exchange Editor is used when integrating several independent models into one simulator without wanting to do the integration explicitly in (model) source code. It is used to describe which output variables in the datapool should be copied to which input variables in the datapool.
The Parameter Exchange Editor is described in detail in Chapter 8.

2.3.5 SMP2 Editor

The SMP2 Editor is used when creating and maintaining SMP2 catalog(ue) files. The catalog(ue) files are usually part of a EuroSim model file and the SMP2 Editor can be invoked from the Model Editor. The SMP2 Editor can also be used as a stand-alone tool. It is capable of generating source code and include files from the catalog(ue) files. The generated source code can then become part of the simulator model.
The SMP2 Editor is described in detail in Chapter 10.

2.3.6 Schedule Editor

The Schedule Editor is used to define the tasks and the schedule of a model.
The Schedule Editor is described in detail in Chapter 11.

2.3.7 Simulation Controller

The Simulation Controller is used to initially define various simulation definitions and also to execute those definitions during a simulation run. Through the Simulation Controller various Action Editors are available, as well as the Initial Condition Editor.
The Simulation Controller is also used to control the actual simulation. It is described in detail in Chapter 12.

2.3.8 Action Editors

To define various actions (stimuli, recorders, interventions, events), a number of Action Editors are available through the Simulation Controller.
The editors are described in detail in Section 12.13.

2.3.9 Initial Condition Editor

With the Initial Condition Editor, initial conditions can be created and modified. An initial condition is used to initialize the simulator, by providing the simulation variables with initial values. The Initial Condition Editor is described in Section 12.6.

2.3.10 Test Analyzer

The Test Analyzer can be used to view and plot the results from a simulation run. Chapter 13, contains more information on the Test Analyzer.

2.4 Facility and project management

With respect to facility and project management, there are the following concepts:

2.4.1 Facility manager

This is the system administrator (having ‘root’ privileges), responsible for the EuroSim installation, including the default (system wide) project file. For more information on the facility managers role, refer to [OM05].

2.4.2 Project file

This is a file holding the definition of a number of EuroSim projects. The default project file is maintained by the user. The project file is located by default in the `.eurosim` directory in the home directory of the user. The location can be changed by defining the `$EFO_HOME` variable. To use a shared project file, a user has to set the `$EFO_HOME` environment variable to point to a shared project file.

2.4.3 Project

A EuroSim project consists of:

- a description
- a directory where the files reside (also called the project root)
- a repository where the versioned files reside
- a version control system name

All this information is stored in the project database.

2.5 Application Programmers Interface

The name *Application Programmers Interface* (API) is used within EuroSim to describe the interface between the model and the EuroSim software. This description includes the services available through EuroSim as well as the variables and functions from the simulation model which need to be accessed by EuroSim.
The API for the EuroSim services is relatively simple: it consists of a number of predefined function calls that can be used from within the user's model code. See Appendix D for a description of the available functions.
The API for the simulation model is a bit more complicated, as EuroSim does not know beforehand what the user's model code will look like. Therefore, in order for the model code to be used in EuroSim, the user has to add API information to the model code: the API *header*. This API header consists of a number of lines at the top of the model code. As the information is stored as comments, the source code will still be usable outside of EuroSim. Using the Model Editor of EuroSim (see Chapter 6), the user can easily enter the functions and variables in the source code which need to be available to EuroSim.
The information from all the API headers in the model together forms the data dictionary of the model (see Section 2.2.3).
The API information required by EuroSim is defined using four keywords (the `'` is part of the keyword):

- `'Global_Input_Variables`
- `'Global_Output_Variables`
- `'Global_State_Variables`
- `'Entry_Point`

The choice of these keywords stems from systems theory, a discipline closely related to the application areas of EuroSim. In systems theory, a classical way to look at systems is from a causal input/output point of view, often referred to as the 'black box' approach to modeling of systems. Inputs are converted to outputs via a so-called black box (Figure 2.3).

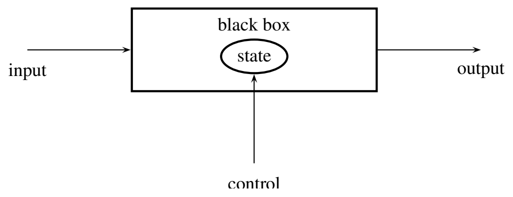


Figure 2.3: The black box approach

An example would be a heater: a current (in Amperes) goes in, a heat flow (in Joules/second) comes out. These inputs and outputs are mapped onto the API-header keywords `'Global_Input_Variables` and `'Global_Output_Variables`.
The next step in the modeling process is to extract (i.e. to model) the memory function of the system. The memory at a certain time is known as the *state* of the system. The state of the system describes in detail how inputs are converted to outputs. Whereas inputs and outputs are the means with which a system communicates to the outside world, there does not exist something like a unique state: the notion of state is very much a mathematical modeling tool.
However, as the system has to be implemented in software to be usable in EuroSim, some way has to be found to define this state. The memory portion of the state is defined using so-called *state variables*. These map onto the keyword `'Global_State_Variables`. The part of the state that determines exactly how to transform input to output using the current state is defined by the functions (or subroutines, or procedures) in the source code. EuroSim assumes that one source code file (i.e. C, Fortran or Ada-95 file) contains one black box.

Note: as far as EuroSim is concerned, it doesn't really matter whether a variable is tagged input, output or state. Each tag will allow EuroSim to access the variable during the simulation. There's only one case where it does make a difference, and that's for the Schedule Editor. This editor can check for data overlap between two tasks, but it will only consider the input and output variables of the tasks' entrypoints in this check.
As EuroSim needs a way to "run" the black box (i.e. to trigger it at the right times) there is a need for a certain amount of control on the black box. This control is given to EuroSim by declaring a number of functions to be an `'Entry_Point`, which means that these functions can be called by EuroSim when necessary.
An additional bonus of specifying all the variables is that it allows the user define some additional attributes, such as description, unit, etc., which might be useful to the Test Conductor and Observer when running the simulator. Also, the variables can be monitored, recorded, or changed during a simulation run if they are defined in the API header.
There are a number of constraints on the model code in order for this API information to be used correctly. First of all, within EuroSim only C, Fortran or Ada-95[2] can be used as languages to build the model. Further, programming language specific, constraints are described in Appendix H.

2.6 Version management

Developing a EuroSim simulation is a continuously moving process. Files are frequently being changed and updated. Especially when more than one person is involved at any one time, it can be difficult to keep track of different versions of a model. In order to assist the user, EuroSim has a number of *version management* facilities built in.
Each of the files used within a simulation can be versioned by the user. Each version of a file can be given an annotation (a short description of the file). Versions are identified by a version number.
When a file is versioned, a *requirement* on that file can be specified: if EuroSim needs access to that file (i.e. when compiling a source file) it then requires a specific version of that file. This could mean that EuroSim needs a version of a file which has since been updated. Therefore a history of the file version is maintained by EuroSim (for versioned files only). For files which are still under development, no requirement should be set. On the other hand, for files that need to be in a stable or predictable state, a version requirement could be used.
The *repository* is the top of a central directory tree where all versions of files for a project are stored[3]. This location is defined when creating a new project (see Section 5.2.3). The project root (which is also defined when creating a new project) contains the *current (working) version* of the files being used for the simulation. When a group of users is accessing the model through the same project directory, they are all working with the same current version. If each user has a project description file of his/her own, or if *tilde expansion* is used for the project root (using the ˜ in a path to represent the users home directory), more than one project root can be defined, which effectively gives each user a private version of the model files.
A copy of any version can be modified at will (e.g. adding new files, or changing existing ones), and when it is decided that a specific file is as it should be, it can be brought under version management by creating a new version. This new version is then the new requirement for the file. Other users can either update their model (by changing the file requirement) or keep using an older version.
Note that *all* files that can be saved from within EuroSim can be put under version management. This includes the simulation model itself, which contains the requirements on the other files. By versioning a model file, a simulation model can be *baselined*, i.e. it can be frozen as a "working simulation".
By versioning all files used for a simulation run, the simulation can be made traceable or reproducible: at any given point in time the simulation can be re-run to recreate simulation results, as the exact version of the model, schedule, initial condition, etc. are stored in the repository.

[2]Note that EuroSim currently only supports creation of the API headers for C and Fortran code. For Ada-95 code, the user should create the API header by hand. See appendix G, API *header layout* for more information on the details of the API header.
[3]Actually, storage is more efficient: only differences of a file with the previous version are stored.

Although the repository can be stored in the same location as the project root, when more than one person is working on a simulation, it is best to keep the repository separate from the project root, so that more than one person can share the same repository, but also keep their own work version.
All versioning actions are done through the *Tools:Version* menu (see Section 3.5.4).
If an existing software repository, created using the RCS or CVS tool, is to be used within EuroSim, this can be accomplished by setting the 'Repository' to the RCS or CVSROOT directory. The 'Project root' should point to an appropriate working directory, with the restriction that the RCS or CVS repository tree has the same structure as the project tree.

Chapter 3

The EuroSim GUI

EuroSim uses a graphical user interface (GUI) for all tools available to the user. This chapter describes the following elements of the user interface:

- Some of the conventions used throughout the user interface.
- The keyboard shortcuts which can be used to quickly access functions from the menus.
- The menu items that are available in every tool.

3.1 GUI conventions in EuroSim

- An ellipsis is shown after a menu item description when a dialog box is shown to request more information from the user, before an action is performed. E.g. *File:Save As...*
- Menu items and buttons that can not be selected (either due to the context, or because they are currently not implemented in EuroSim) are shown grayed out.
- Where applicable, keyboard shortcuts are shown next to the item. For more information, refer to Section 3.3.

As the EuroSim GUI's are based upon the Qt toolkit, the following elements are used for user input:

- *Checkboxes* (little squares) which can be selected by pressing the box.
- *Radiobuttons* (circles) which behave the same as checkboxes, with the exception that of a group of related radiobuttons, only one can be active.
- *Normal buttons* (rectangles), which have a descriptive label such as 'Save' on top of the button. Pushing the button performs an action.
- *Textfields* (large rectangular areas, sometimes with sliders alongside it), which can be used to enter text. If the field has sliders, they can be used to reveal parts of the field which are not shown on screen.

3.2 Mouse buttons

An item in a window is selected by placing the mouse pointer over it and clicking the left mouse button (MB1). More objects can be selected by holding down the Control or Shift key when clicking MB1. Double-clicking an item with MB1 will activate it (i.e. do the thing the icon represents, e.g. drawing a plot) or fold/unfold it, in case it is an icon in a tree structure.
Pressing the left mouse button over a selected icon allows one to drag the icon and drop it somewhere else (e.g. in a monitor definition, that will then be extended with the new variable name).

3.3 Keyboard shortcuts

The menu items can also be accessed using the keyboard. There are two methods:

- The Alt key can be used to access the menubar. Once selected, menu options can be selected by using the cursor keys followed by Return or by typing the underlined letter for a particular menu option. Escape aborts from the menu traversal.
- Specific, often used, menu items can also be selected directly using a short cut. These shortcuts are usually combinations of the Ctrl and Alt keys and a character key, and are shown next to the menu item.

In textfields, the usual editing keys such as Tab, Enter, arrow keys, Home and End are available. Besides these keys, the following keys have special meaning:

- Prior (or PageUp) scrolls down a page
- Next (or PageDown) scrolls up a page
- Ctrl+a moves to the beginning of the line
- Ctrl+b moves the cursor backwards a character
- Ctrl+c copies the selected text to the clipboard
- Ctrl+d deletes a character
- Ctrl+e goes to the end of the line
- Ctrl+f moves the cursor forward a character
- Ctrl+h backspaces a characters
- Ctrl+k deletes to the end of the line, or removes an empty line
- Ctrl+n moves to the next line
- Ctrl+p moves to the previous line
- Ctrl+v inserts text previously cut or copied
- Ctrl+x cuts selected text from the field
- F2 starts editing a selected label in a tree view

On systems running the X Window System (UNIX platforms), the second mousebutton inserts the Xbuffer selection at the cursor location.

3.4 Common buttons

There are a number of buttons that are used throughout EuroSim.

OK	Acknowledges the question, or accept the changes made in a window and close the window.
Cancel	Abort the operation and all entered data is ignored.
Apply	Accept the changes made in a window, but do not close the window.
Dismiss	Close the dialog window.
Browse	Open a dialog to select an item from a list. Often used to select a file.

3.5 Common menu items

Throughout EuroSim, a number of menus appear with every tool. These menus have a number of 'standard' items, which are described in this section. Note that each tool can add a number of tool-specific items to these menus - these tool-specific items are described in the sections on these tools.

3.5.1 File menu

New	A new file will be created. If there are any unsaved changes in the current file, a warning dialog box will pop up and ask whether you want to save the changes first.
Open	Pop-up a file selection dialog box in which a file to be opened can be selected. If there are any unsaved changes to the current file, first a warning dialog box will appear (see *New*).
Save	Save the current file without closing it. If the current file has never been saved before (an 'Untitled' file), a file selection dialog box will pop-up asking the user to enter the name of the file. Note that this item cannot be selected if there are no unsaved changes. Note that a window title will have an asterisk appended to the name of the file in the title if the file needs to be saved.
Save As	Save the current file with a different name. The newly created file will become the current file.
Print	Print the current file in an appropriate form.
Exit	Close the tool and all windows associated with it. If there are any unsaved changes, a warning dialog box will pop up.

3.5.2 Edit menu

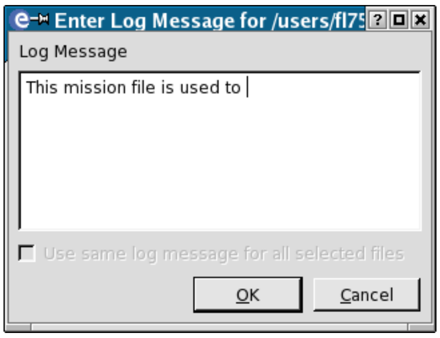
Undo	Undo the last action performed by the user.
Redo	Redo the last undone action.
Cut	Move the selected portion of data from the tool window to the clipboard.
Copy	Copy the selected portion of data from the tool window to the clipboard.
Paste	Move the contents of the clipboard to the tool window. Depending on the tool, the location where to paste can be selected.
Delete	Remove the selected portion of data from the tool window.

3.5.3 Tools menu

Shell	Start a command line session (also known as 'xterm' on X Window Systems (UNIX platforms), or 'Command Prompt' on Windows NT platforms).

3.5.4 Tools:Version menu

Add...	Add the selected file to the repository. A dialog appears where you can enter a text describing the change. See Figure 3.1 for an example.

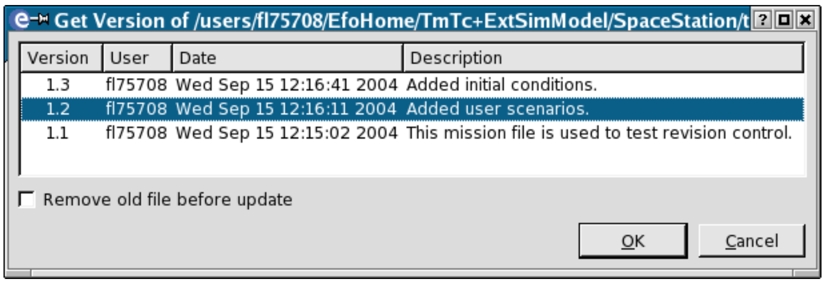

Enter Log Message for /users/fl75
Log Message
Use same log message for all selected files
OK
Cancel

```
This mission file is used to
```

Figure 3.1: The Log Message

Update Update the selected file with the latest version from the repository.

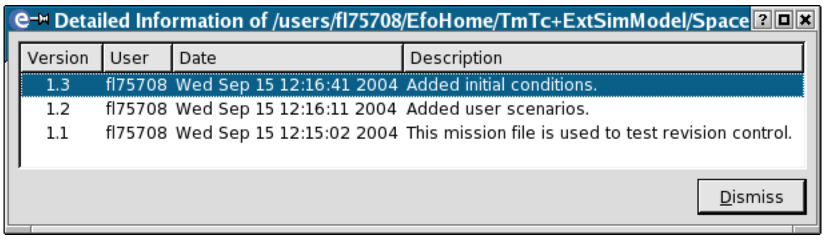
Get... Get a specific version of the selected file from the repository. If the checkbox **Remove file before update** is checked, then before the selected version is retrieved, the old file is removed. Otherwise the selected version is merged with the current version. The version with a checkmark in front is the required version.


Get Version of /users/fl75708/EfoHome/TmTc+ExtSimModel/SpaceStation/t
Remove old file before update
OK
Cancel

Version	User	Date	Description
1.3	fl75708	Wed Sep 15 12:16:41 2004	Added initial conditions.
1.2	fl75708	Wed Sep 15 12:16:11 2004	Added user scenarios.
1.1	fl75708	Wed Sep 15 12:15:02 2004	This mission file is used to test revision control.

Figure 3.2: Get Version

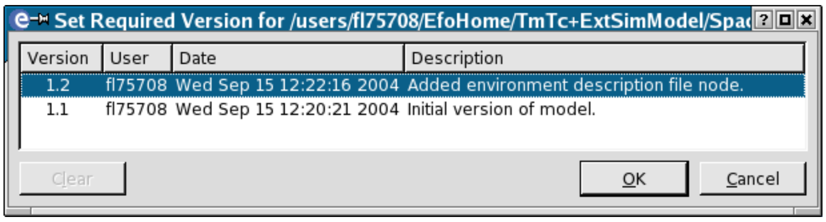
Detailed...
Show the detailed version history of the selected file. The version with a checkmark in front is the required version.


Detailed Information of /users/fl75708/EfoHome/TmTc+ExtSimModel/Space
Dismiss

Version	User	Date	Description
1.3	fl75708	Wed Sep 15 12:16:41 2004	Added initial conditions.
1.2	fl75708	Wed Sep 15 12:16:11 2004	Added user scenarios.
1.1	fl75708	Wed Sep 15 12:15:02 2004	This mission file is used to test revision control.

Figure 3.3: Detailed Information

Set Required...
Select a required version of the selected file. The version with a checkmark in front is the current required version.

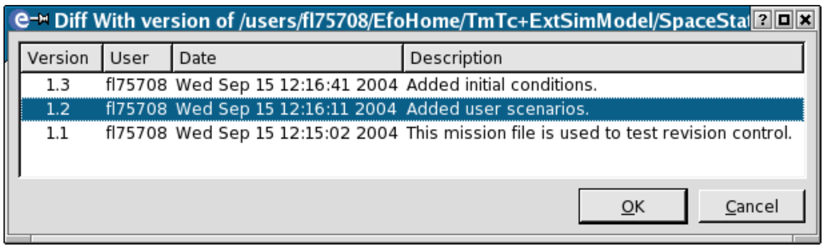

Set Required Version for /users/fl75708/EfoHome/TmTc+ExtSimModel/Spac
Clear
OK
Cancel

Version	User	Date	Description
1.2	fl75708	Wed Sep 15 12:22:16 2004	Added environment description file node.
1.1	fl75708	Wed Sep 15 12:20:21 2004	Initial version of model.

Figure 3.4: Set Required Version

Diff with. . .

Show the differences of the selected file with another version of that file. The version with a checkmark in front is the required version.


Diff With version of /users/fl75708/EfoHome/TmTc+ExtSimModel/SpaceSta
OK
Cancel

Version	User	Date	Description
1.3	fl75708	Wed Sep 15 12:16:41 2004	Added initial conditions.
1.2	fl75708	Wed Sep 15 12:16:11 2004	Added user scenarios.
1.1	fl75708	Wed Sep 15 12:15:02 2004	This mission file is used to test revision control.

Figure 3.5: Difference With

3.5.5 Help menu

Online Help. . .

Provide a short description of the tool.

About EuroSim

Show the version of EuroSim.

Chapter 4

EuroSim tutorial

In this chapter, a complete pass through the EuroSim life-cycle is described. An example is used to describe all steps necessary to create a successful simulation with EuroSim. The user is advised to check the reference part of the user manual (Chapter 5, and onwards) for more information on menu items and the various objects in the EuroSim environment. See Section 2.1 for a description of the life-cycle.

4.1 The case study

Throughout this user guide, a complete ready-to-run simulator is developed. A simple model of a satellite that hovers above a planet, without having it in a geostationary orbit, is used. The altitude of the satellite decays by perturbations and by the gravity pulling it to the planet surface. The thruster is switched on when the altitude reaches a lower limit and is switched off when the satellite reaches an upper limit.

4.2 Starting EuroSim

To run EuroSim on a UNIX platform, perform the following steps:

- Only for IRIX:

  Set the environment variable `EFOROOT` to the directory where EuroSim has been installed (by default this is `/usr/EuroSim`);

- Only for IRIX:

  Issue the command `. $EFOROOT/etc/user.sh` (for Bourne shell or compatible) or `source $EFOROOT/etc/user.csh` (for C-shell or compatible);

- Start EuroSim by typing `esim` at the command prompt.

To run EuroSim on a Windows NT platform, select *EuroSim* from *Start Menu:Programs*, or double-click on the EuroSim icon on the desktop.

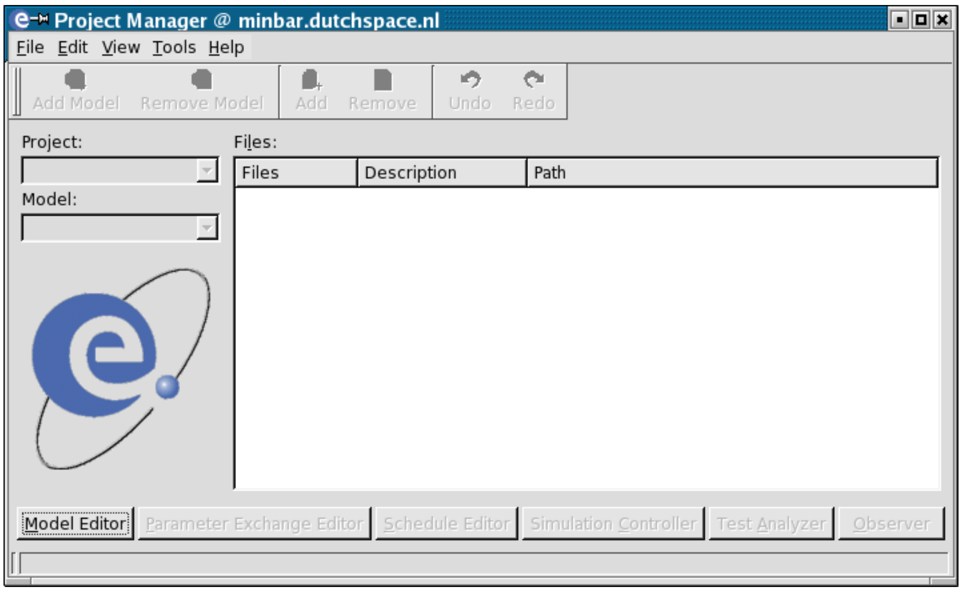


Figure 4.1: The main EuroSim window

After a short while, the main EuroSim window will appear (see Figure 4.1). This window will display the projects to which you have access. If no project is shown ask the EuroSim facility manager to create one for you, or alternatively, create your own project, as described in the next section.

4.3 Creating a project yourself

Select *File:Add Project...* from the menu. To create a new project, enter the project name, choose the project directory and version control system. The 'Description' and 'Repository Root' fields are optional. For the remainder of this chapter, the name '`SUM`' is assumed.

4.4 Creating a shared project

Instead of using a project created by yourself, you can create shared project(s) and database managed by the EuroSim facility manager. This can be achieved by doing the following, *before* starting EuroSim as described in the previous section.

- The EuroSim Facility Manager creates a directory where the shared project database can be stored.
- Set the environment variable `EFO_HOME`[1] to this directory.
- Start EuroSim (see Section 4.2).

4.5 Creating a model

In the main EuroSim window, select the project to be used for this case study from the **Project** combobox and press the **Model Editor** button to create a new model. The Model Editor will show.
When creating a new model a basic model structure consisting of the root node will appear. When editing an existing model select *File:New* to create this basic model structure (see Figure 4.2).

[1]On a Windows NT platform, environment variables are defined in the file `$EFOROOT/bin/esim.bashrc`.

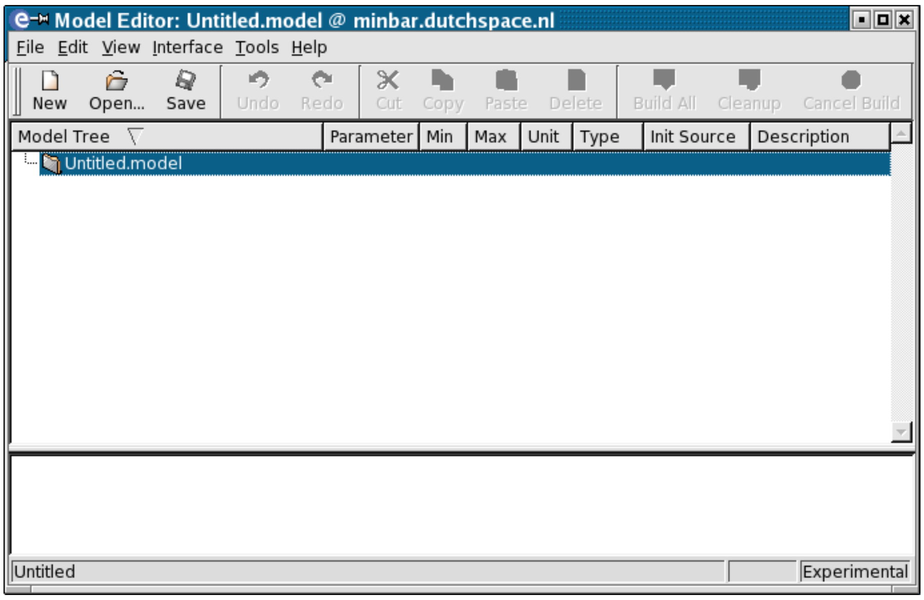

Model Editor: Untitled.model @ minbar.dutchspace.nl
File Edit View Interface Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete Build All Cleanup Cancel Build
Model Tree
Parameter Min Max Unit Type Init Source Description
Untitled.model
Untitled
Experimental

Figure 4.2: A new model

4.5.1 Model

The model for this simulation is divided into four parts:

- a sub-model that decreases the altitude of the satellite;
- a sub-model that lifts the satellite to a higher altitude by usage of a thruster;
- a sub-model that initializes the altitude decay sub-model;
- a sub-model that initializes the thruster sub-model.

The two initialization sub-models will initialize all the variables of the model.
The thruster sub-model will monitor the altitude and keep it within limits. These limits are between 210 km and 280 km respectively. When it is below the lower limit the thruster will increase the altitude until it reaches the upper limit. At that point it will wait until the altitude has decayed to the lower limit and the process starts all over again. In Figure 4.3 the flowcharts of the two main sub-models are shown. These flowcharts could be compared to a first version of the design. Later on in the case study, more optimized code will be used.

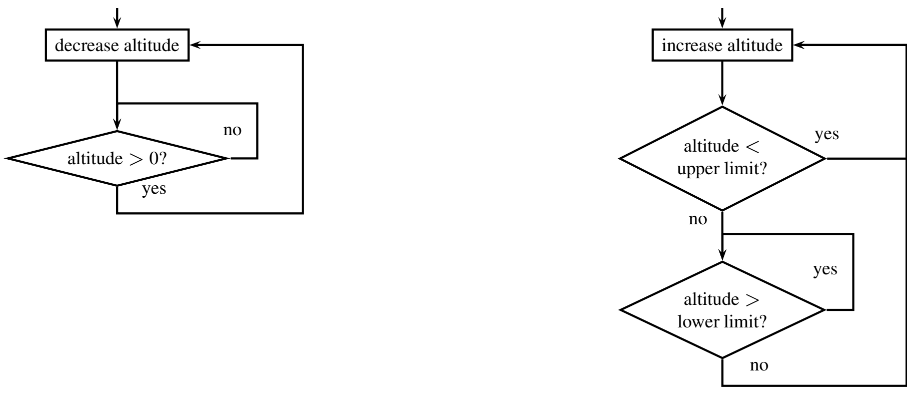


Figure 4.3: The altitude (left) and thruster models

4.5.2 Adding the sub-models

In order to add the four sub-models to the model, select the root node (the left-most node), and choose *Edit:Add Org Node* from the menu. In the window that appears, enter as name `Altitude`. Add another org node (after first selecting the root node again, if necessary), and this time use the name `Thruster`.
The next level of the model hierarchy will consist of four source files, each corresponding to one of the four sub-models. Start by selecting the 'Altitude' node and then do an *Edit:Add File Node*. In the window that appears, enter as file name `Initialize_Altitude.f`, or use the file selection dialog if you already have the tutorial source files. EuroSim will recognize this file as a Fortran source file. A new file node will be added to the model hierarchy.
Repeat the process for the three other file nodes: attach a file node with file name `Altitude.f` to the `Altitude` node, and add two file nodes with names `Initialize_Thruster` and `Thruster` respectively to the `Thruster` node (using files `Initialize_Thruster.c` and `Thruster.c`).
By now, the model should look like Figure 4.4. Notice that after making changes to the new model, as asterisk ($\star$) is shown in the title bar of the window to indicate that there are changes to be saved.

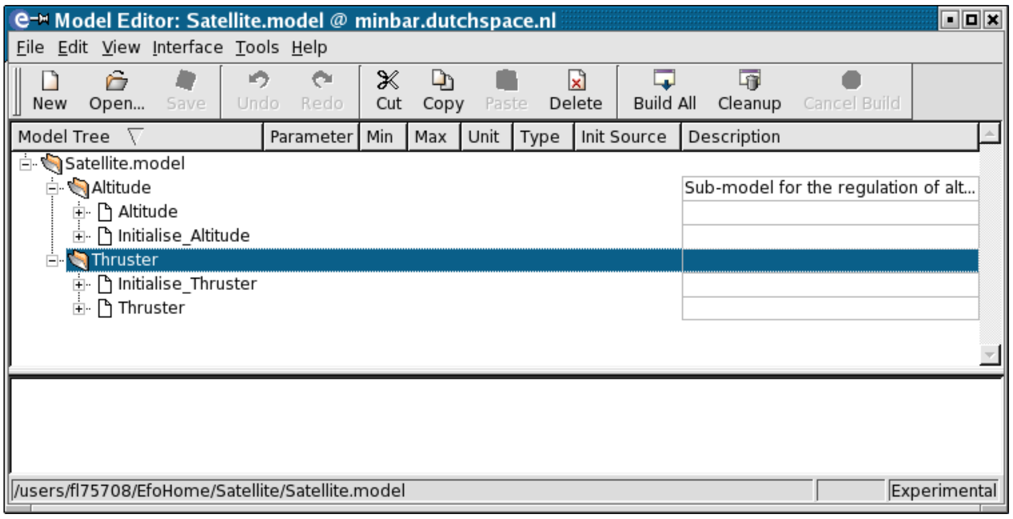

Model Editor: Satellite.model @ minbar.dutchspace.nl
File Edit View Interface Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete Build All Cleanup Cancel Build
Model Tree Parameter Min Max Unit Type Init Source Description
Satellite.model
Altitude
Altitude
Initialise_Altitude
Thruster
Initialise_Thruster
Thruster
Sub-model for the regulation of alt...
/users/fl75708/EfoHome/Satellite/Satellite.model
Experimental

Figure 4.4: Model with the file nodes

Save the model by selecting *File:Save*. As model name, enter `SUM.model` in the file selection window. This file selection is shown because the new model has not been saved before. The next time the model is saved, no file selection window is shown.

4.5.3 Adding the source code

Next, the actual source files have to be created[2]. Do this by selecting the `Altitude` file node, and choosing *Edit:Edit Source* from the menu. An editor[3] will show, in which the following source code should be entered. Beware that Fortran wants to have 6 spaces before the first character on the line (except for the comment lines starting with 'C' in column 1). This is a left-over from the times that programs were entered using punch cards.

Listing 4.1: Source `Altitude.f`

```
C---------------------------------------------------------
C File: Altitude.f
C
C Contents: The Fortran routines that simulate the gravity
C pull of a planet.
C
C---------------------------------------------------------
```

[2] If the files have already been selected with the file selection dialog, this step can be skipped.
[3] See Section 6.5 for information on how to change the default editor.

```
      SUBROUTINE DECAYALTITUDE

C     Global Variable definition.
      INTEGER ALTITUDE
      INTEGER DECAYSPEED, DECAYCOUNTER

C     COMMON Block Definition.
      COMMON /ALTDATA/ ALTITUDE, DECAYSPEED, DECAYCOUNTER

      DECAYCOUNTER = DECAYCOUNTER + 1
      IF (DECAYCOUNTER .GT. DECAYSPEED) THEN
        DECAYCOUNTER = 0
        IF (ALTITUDE .GT. 0) THEN
          ALTITUDE = ALTITUDE - 1
        ENDIF
      ENDIF

      RETURN
      END
```

Save the source file, and close the editor. Repeat the process for `Initialize_Altitude` with the source file:

Listing 4.2: Source `Initialize_Altitude.f`

```
C--------------------------------------------------------------------
C File: Initialize_Altitude.f
C
C Contents: Initialize the altitude decay simulation model.
C
C--------------------------------------------------------------------
      SUBROUTINE INITIALIZEALTITUDE

C     Global Variable definition.
      INTEGER ALTITUDE
      INTEGER DECAYSPEED, DECAYCOUNTER

C     COMMON Block Definition.
      COMMON /ALTDATA/ ALTITUDE, DECAYSPEED, DECAYCOUNTER

C     Parameter Definition.
      PARAMETER (DECAYSPEEDDEFAULT = 100)

      ALTITUDE = 0
      DECAYCOUNTER = 0
      DECAYSPEED = DECAYSPEEDDEFAULT

      RETURN
      END
```

Listing 4.3: The C source code for the `Thruster` file node

```
/*
 File: Thruster.c

 Contents: The C routines that simulate the thruster module
 of the satellite.
*/

#define On       1
#define Off      0
```

```
extern int altitude;
int thrusterOnOff;
int speedCounter = 0;
int satelliteAscentSpeed;
int lowerAltitudeLimit;
int upperAltitudeLimit;

void Thruster(void)
{
  if (thrusterOnOff == On) {
    if (speedCounter++ > satelliteAscentSpeed) {
      speedCounter = 0;
      altitude++;
      thrusterOnOff = (altitude < upperAltitudeLimit);
    }
  }
  else {
    thrusterOnOff = (altitude < lowerAltitudeLimit);
 }
}
```

Listing 4.4: The source file for the `Initialize_Thruster` node

```
/*
  File: Initialize_Thruster.
  Contents: Initialize the thruster simulation model.
*/

#define SPEED_DEFAULT 10
#define On        1
#define Off       0

extern int speedCounter;
extern int satelliteAscentSpeed;
extern int thrusterOnOff;
extern int lowerAltitudeLimit;
extern int upperAltitudeLimit;

void Initialize_Thruster(void)
{
  satelliteAscentSpeed = SPEED_DEFAULT;
  speedCounter = 0;
  thrusterOnOff = On;
  lowerAltitudeLimit = 210;
  upperAltitudeLimit = 280;
}
```

4.5.4 Adding the API headers

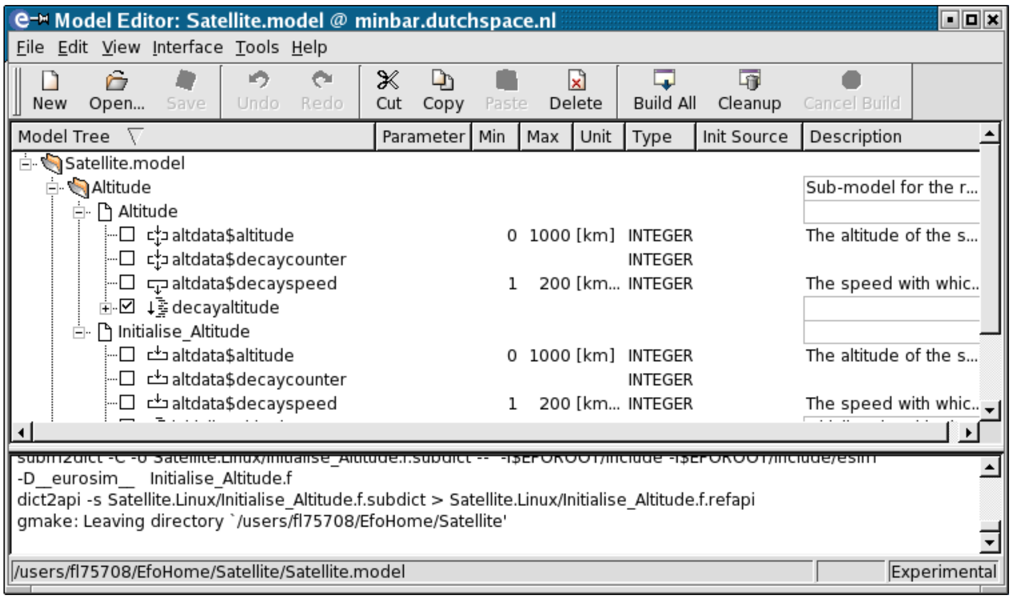
4.5.4.1 The Altitude sub-model

The next step is to add the API headers to the model. Expand the `Altitude` file node by pressing the '+' symbol, or use *View:Expand All*. EuroSim will parse the expanded file(s) and display the available entries and variables in the code. First, the `decayaltitude` entrypoint will be added to the API header. Click the checkbox left to `decayaltitude` to add this entrypoint to the API header.
We will also add two of the variables from this entrypoint to the API header: tick the checkboxes in front of the `altdata$altitude` and `altdata$decayspeed` variables under the `decayaltitude` entrypoint.

When added to the API header (checkmark used), additional information on entrypoints and variables can be entered (such as a description). Select the `decayaltitude` entrypoint and click the 'Description' field on the right. Enter the description `The altitude decay operation`. Select the `altdata$altitude` variable. The 'Type' and 'Init Source' fields cannot be changed, as they are extracted from the source file. Enter a description of `The altitude of the satellite`. Enter as 'Unit' the string `[km]`, as 'Min' the value 0 and as 'Max' the value 1000. Repeat this for the `altdata$decayspeed` variable, using the values:

Description	`The speed with which the altitude decays`
Unit	`[km/s]`
Min	1
Max	200

The model should now look like Figure 4.5.
Repeat the above steps for the three remaining sub-models, using the values from the next sections.


Model Editor: Satellite.model @ minbar.dutchspace.nl
File Edit View Interface Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete Build All Cleanup Cancel Build

Model Tree	Parameter	Min	Max	Unit	Type	Init Source	Description
Satellite.model							
Altitude							Sub-model for the r...
Altitude							
altdata$altitude		0	1000	[km]	INTEGER		The altitude of the s...
altdata$decaycounter					INTEGER		
altdata$decayspeed		1	200	[km...	INTEGER		The speed with whic..
☑ decayaltitude							
Initialise_Altitude							
altdata$altitude		0	1000	[km]	INTEGER		The altitude of the s...
altdata$decaycounter					INTEGER		
altdata$decayspeed		1	200	[km...	INTEGER		The speed with whic..

```
subm2dict -C -o Satellite.Linux/Initialise_Altitude.f.subdict --  -I$EFOROOT/include -I$EFOROOT/include/esim
-D__eurosim__   Initialise_Altitude.f
dict2api -s Satellite.Linux/Initialise_Altitude.f.subdict > Satellite.Linux/Initialise_Altitude.f.refapi
gmake: Leaving directory `/users/fl75708/EfoHome/Satellite'
```

/users/fl75708/EfoHome/Satellite/Satellite.model Experimental

Figure 4.5: The expanded Altitude node

4.5.4.2 The `Initialize_Altitude` sub-model

Add the entrypoint in `initializealtitude` with a description `Initialize the altitude decay operations.`

4.5.4.3 The Thruster sub-model

Add the entrypoint `Thruster` with a description `The thruster brings the satellite to the correct altitude.` Add the following variables by selecting them from the list to the right of the `Thruster` entrypoint:

Variable	**Min**	**Max**	**Unit**	**Description**
`lowerAltitudeLimit`	0	1000	`[km]`	`Below this limit, thruster must be turned on`
`satelliteAscendSpeed`	1	200	`[km/s]`	`The ascent speed of the satellite`
`thrusterOnOff`	0	1	`[1=On/0=Off]`	`Thruster on/off indicator`
`upperAltitudeLimit`	0	1000	`[km]`	`Above this limit,thruster must be turned off`

4.5.4.4 The `Initialize_Thruster` sub-model

Add the entrypoint `Initialize_Thruster` with a description `Initialize the thruster.`

4.6 Building the simulator

Select *Tools:Build All* from the menu in the Model Editor. In the output window, all commands executed are echoed, as well as their outputs. Things to look out for are lines starting with `*** Error`, which indicate that an error has occurred during building. Usually directly above a more descriptive error message is given. You can ignore the file version warnings, but there should be an error message like:

```
Satellite.Linux/Thruster.pub.o: In function 'Thruster':
Satellite.Linux/Thruster.pub.o(.text+0x2b): undefined reference to 'altitude'
Satellite.Linux/Thruster.pub.o(.text+0x31): undefined reference to 'altitude'
Satellite.Linux/Thruster.pub.o(.text+0x4e): undefined reference to 'altitude'
collect2: ld returned 1 exit status
gmake: Leaving directory '/home/jv75763/work/Satellite'
gmake: *** [Satellite.Linux/Satellite.exe] Error 1
*** Errors during build ***
```

The meaning of this message is that the compiler can not find a declaration with the name `altitude`. Inspection of the source files indicates that the C function `Thruster` uses an external declaration of a variable with the name `altitude`. Although the Fortran source has a variable with the name `ALTITUDE` it is not possible to connect these two variables in the way the current satellite model has been written. This is a general problem with linking Fortran and C code. It arises from compiler conventions, not from the EuroSim tools.
To solve the problem, change the `altitude` variable in the file `Thruster.c` to the following `struct` declaration:

```
extern struct altitudeDataStruct
{
 int ALTITUDE;
 int DECAYSPEED;
 int DECAYCOUNTER;
} altdata_;
```

And change the use of the variable `altitude` to:

```
altdata_.ALTITUDE
```

Note that the `altitude` variable is used in three places. Be sure to change them all. The `Thruster.c` source file should now look like:

```
/*
  File: Thruster.c

  Contents: The C routines that simulate the thruster module
  of the satellite.
*/

#define On       1
#define Off      0

extern struct altitudeDataStruct
{
  int ALTITUDE;
  int DECAYSPEED;
  int DECAYCOUNTER;
} altdata_;
```

```
int thrusterOnOff;
int speedCounter = 0;
int satelliteAscentSpeed;
int lowerAltitudeLimit;
int upperAltitudeLimit;

void Thruster(void)
{
  if (thrusterOnOff == On) {
    if (speedCounter++ > satelliteAscentSpeed)  {
      speedCounter = 0;
      altdata_.ALTITUDE++;
      thrusterOnOff = (altdata_.ALTITUDE < upperAltitudeLimit);
    }
  }
  else {
    thrusterOnOff = (altdata_.ALTITUDE < lowerAltitudeLimit);
  }
}
```

When the changes to the source file have been made, try rebuilding the simulator. If the build was successful, the messages `SUM.exe MADE` and `all DONE` should be displayed in the status window.
Save the model and exit the model editor. In the EuroSim main window choose *Edit:Add Model* and select `SUM.model` to add the created model to the project.

4.7 Creating the schedule

The schedule of a simulation defines which tasks need to be activated at which time. A task is a set of entrypoints which are executed sequentially. Task and schedule can be created using the Schedule Editor. Select the EuroSim main window and press the 'Schedule Editor' button.
The schedule contains four tab pages, one for each of the simulator states *initializing*, *executing*, *standby* and *exit*. For the example, three of the four states will be used.
In the initializing state, a schedule will be created which will be triggered by state entry, and which will then initialize the thruster and altitude model. After these have been executed, the schedule will put the simulator in standby state.
For the executing state, a schedule will be created which triggers the thruster and altitude models using two timers, one at 20 Hz and one at 100 Hz.
In the exit state, a schedule will be created which will close down the simulator.

4.7.1 Initializing schedule

Choose *File:Select Model* from the menu. Select the file `SUM.model` to be able to use the created API header.
Select the circle symbol from the toolbar for a task[4]. The cursor changes into a circle. Put the circle on the schedule tab page. It will change color to red, indicating an error (in this case: the task has no input and output connectors attached). It will get a default name of `New Task`. Select the arrow tool from the toolbar on the left. Double click on the task, which causes the task properties dialog to open. In this dialog, select the `Initialize_Thruster` entrypoint on the left Data Dictionary view and press the **Add** button. This will copy the entrypoint to the Entrypoints list, indicating that this entrypoint belongs to the task we are defining. Do the same with the `Initialize_Altitude` entrypoint.
When a task is executed, each of the entrypoints contained in the task will be executed sequentially. For this initializing task the order is not important, but if it is, the up and down arrow buttons can be used to re-order the entrypoints. Timing information can be entered for each entrypoint. As we don't have such information at this moment, we will leave it empty. Later on, if the simulation has been executed

[4]See Section 11.2 for a description of which icon belongs to which item.

successfully, it is possible to import a timings file created by the simulator, which contains the various data required here.
Now change the name of the task to `Initialize` by entering the new name in the field Taskname below the Data Dictionary box. Press the **OK** button. The task on the Schedule Editor now also has the name `Initialize`.
Next, from the *Insert* menu, select the menu item *Internal Event*. Select `STATE_ENTRY` from the sub-menu. Put it on the tab page. Next select a flow (curved arrow) from the tool button bar. Click the left mouse button on the internal event. Keep the left mouse button pressed and move the mouse to the task. Notice how the flow follows the cursor. Release the left mouse button again above the task. The two are now connected.
Finally, add the `PAUSE` output connector to the tab page, and connect a flow from the task to the output connector. The initializing schedule should now look something like Figure 4.6.

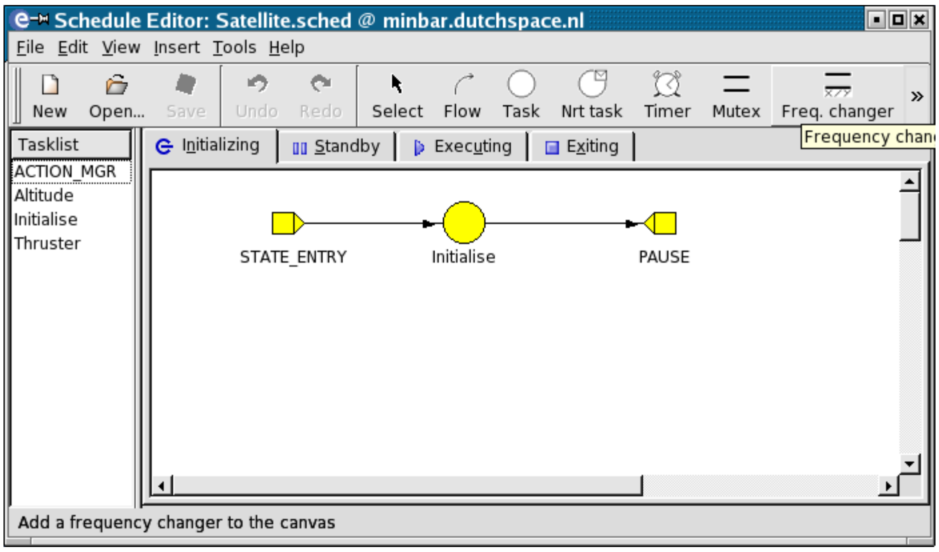

Schedule Editor: Satellite.sched @ minbar.dutchspace.nl
File Edit View Insert Tools Help
New Open... Save Undo Redo Select Flow Task Nrt task Timer Mutex Freq. changer
Frequency chan
Tasklist
ACTION_MGR
Altitude
Initialise
Thruster
Initializing Standby Executing Exiting
STATE_ENTRY
Initialise
PAUSE
Add a frequency changer to the canvas

Figure 4.6: The initializing schedule

4.7.2 Executing schedule

First select the *Executing* tab to show the schedule for the executing state. On the tab page, create two more tasks, named `Thruster` and `Altitude`. The `Thruster` task should contain the `Thruster` entrypoint, and the `Altitude` task should contain the `decayaltitude` entrypoint.
Next to each task, put a timer. Connect each timer to a task using a flow. As the `Altitude` task should be executed less often than the `Thruster` task, double-click on the timer connected to the `Altitude` task. A timer attribute window will show. In the window, change the frequency to 20 Hz. Close the window with the **OK** button.
Change the frequency of the `Thruster` timer to 100 Hz. On some operating systems this is the default frequency. Other operating systems may have a different default frequency setting.
The executing schedule should now look something like Figure 4.7. With this schedule, the `Thruster` task will be triggered with a frequency of 100 Hz, and the `Altitude` task with a frequency of 20 Hz.

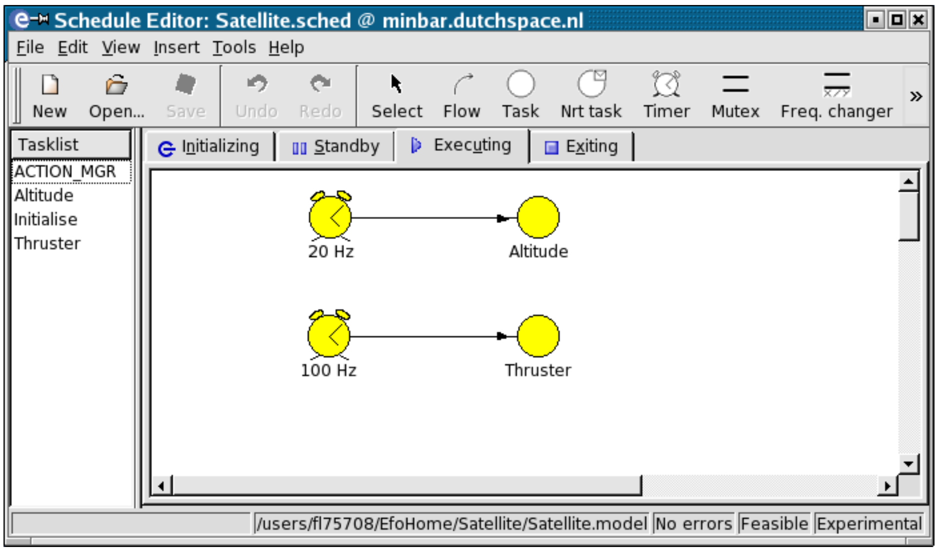


Figure 4.7: The executing schedule

4.7.3 Closing the Schedule Editor

After each of the schedules has been created, select *File:Exit* from the menu and select **Save** when a warning is given about unsaved changes. In the Model Editor, save the model.

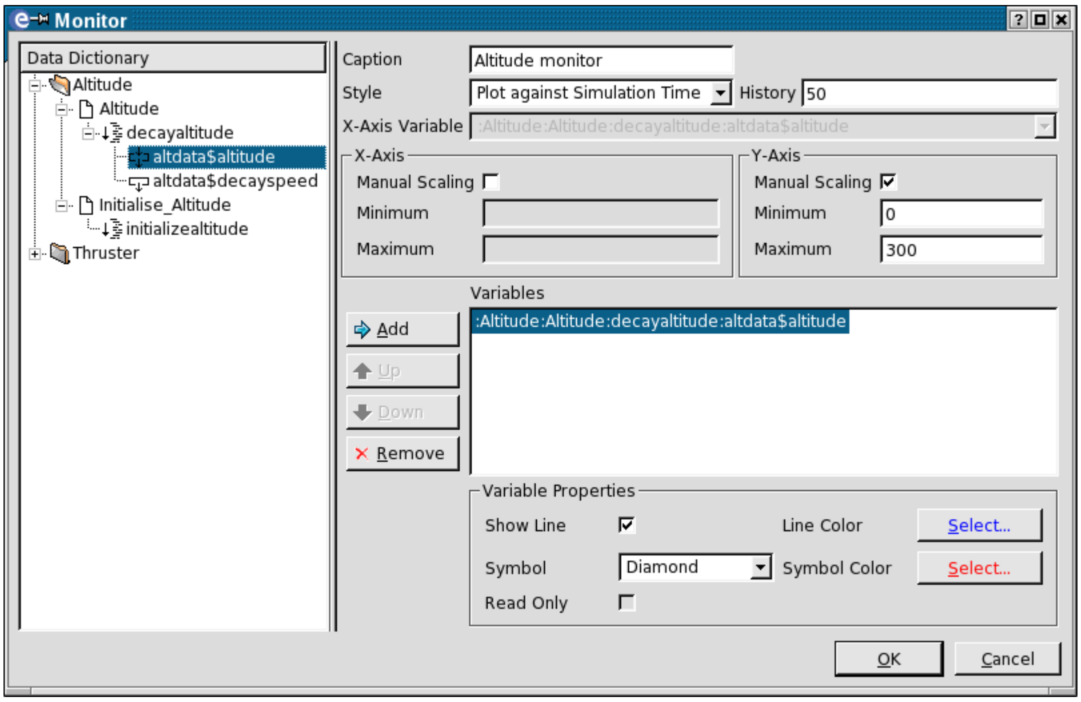
4.8 Creating a simulation definition

Now that the model has been created and the simulator has been built, a simulation definition should be created. A simulation definition contains information on the initial values of the variables defined in the API headers, as well as stimuli, recorders and monitors, which can be used to monitor and influence the simulation.
Select **Simulation Controller** from the main EuroSim window. The Simulation Controller will start (see Chapter 12).
In order to create a simulation definition, the Simulation Controller needs to know which particular model and schedule the simulation is intended for (which indirectly gives access to the associated data dictionary). Choose *File:New* to create a new simulation definition. A wizard dialog appears where you can select all files that you want to use in a simulation. Initially you must select the `SUM.model` and the `SUM.sched` files. Use the **Browse…** button to select the model, press the **Next** button to go to the next page of the wizard. If the prefilled schedule file (guessed from the model file) is correct then press **Finish**, otherwise use the **Browse…** button to select the right schedule file and press **Finish**.

4.8.1 Creating a graphical monitor

Select *Insert:New* MMI… from the menu. You are asked to choose a filename for the new Man-Machine Interface file. Save the file as `Altitude.mmi`. Now you will be asked for the caption of the new tab page. By default the name of the file without the suffix will be chosen. Accept the default.
A blank tab page named *Altitude* appears where you can add monitors. Select this tab and choose *Insert: New Monitor* to add a new monitor. The Monitor Editor will appear (see Section 12.2.5 for more information).
In the Monitor Editor, enter `Altitude monitor` as the caption. Now expand the `decayaltitude` node and double click the variable `altdata$altitude` on the Dictionary Browser. The variable appears in the *Variables* list and is now connected to the monitor.
Change the style from 'Alpha Numeric' to 'Plot against Simulation Time'. By default the X and Y axis will scale automatically when the plot is being created. Select 'Manual Scaling' to define the min/max

range yourself. As you can see, the first time you select Manual Scaling the min and max values will be determined from the *Variables* list (if possible). The Monitor Editor should now look like Figure 4.8. Close the editor with the **OK** button. On the Altitude tab page, the new monitor is shown.


Monitor

Data Dictionary
- Altitude
  - Altitude
    - decayaltitude
      - altdata$altitude
      - altdata$decayspeed
  - Initialise_Altitude
    - initializealtitude
- Thruster

Caption	Altitude monitor
Style	Plot against Simulation Time
History	50
X-Axis Variable	:Altitude:Altitude:decayaltitude:altdata$altitude

X-Axis		Y-Axis	
Manual Scaling	☐	Manual Scaling	☑
Minimum		Minimum	0
Maximum		Maximum	300

Variables
:Altitude:Altitude:decayaltitude:altdata$altitude

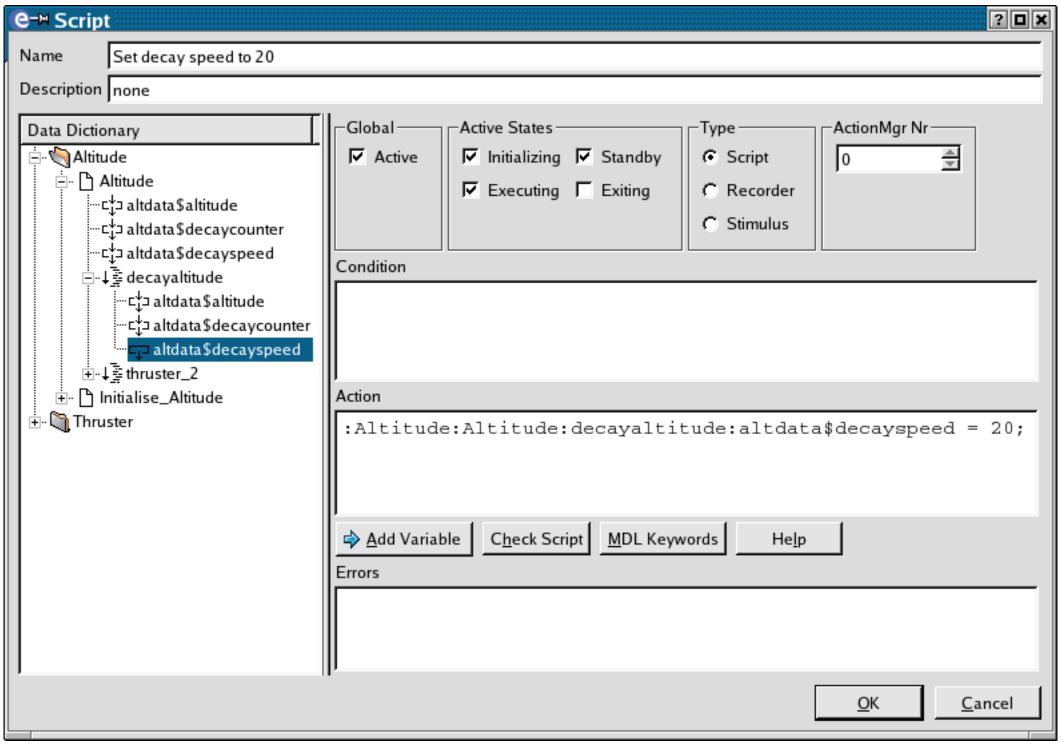
Variable Properties

Show Line	☑	Line Color	Select...
Symbol	Diamond	Symbol Color	Select...
Read Only	☐		

Figure 4.8: The Monitor Editor

4.8.2 Creating an intervening action

In order to create an action which changes a variable during the simulation, you first have to create a scenario file where such actions are defined. Choose *Insert:New Scenario* from the menu. Save the file as `SUM.mdl`. Now you will be asked for the caption of the new tab page. By default the name of the file is used without the suffix. Accept the default.
To add a script choose *Insert:Script* from the menu. Change the name of the action to `Set decay speed to 20`. Select the options 'Initializing' and 'Standby'. Because this action should only be executed if the Test Conductor wants it, the 'Condition' field is left blank. Now the action has to be started explicitly by the Test Conductor.
Select the variable `altdata$decayspeed` from the Dictionary Browser using the left mouse button. Whilst keeping the mouse button pressed, drag the name of the variable to the *Action* field. Release the button. The variable is now copied to the Action field. Add `=20` to the same line as where the variable is shown. This statement means to set the variable to a value of 20. Optionally, press **Check Script** to see if any errors were made. The Script Editor should now look like Figure 4.9.
Close the Script Editor with the **OK** button. The new action appears on the Scenario tab page.


Script

Name: Set decay speed to 20

Description: none

Data Dictionary
- Altitude
  - Altitude
    - altdata$altitude
    - altdata$decaycounter
    - altdata$decayspeed
    - decayaltitude
      - altdata$altitude
      - altdata$decaycounter
      - altdata$decayspeed
    - thruster_2
  - Initialise_Altitude
- Thruster

Global: [✓ Active]

Active States: [✓ Initializing] [✓ Standby] [✓ Executing] Exiting

Type: [✓ Script] Recorder Stimulus

ActionMgr Nr: 0

Condition

Action

```
:Altitude:Altitude:decayaltitude:altdata$decayspeed = 20;
```

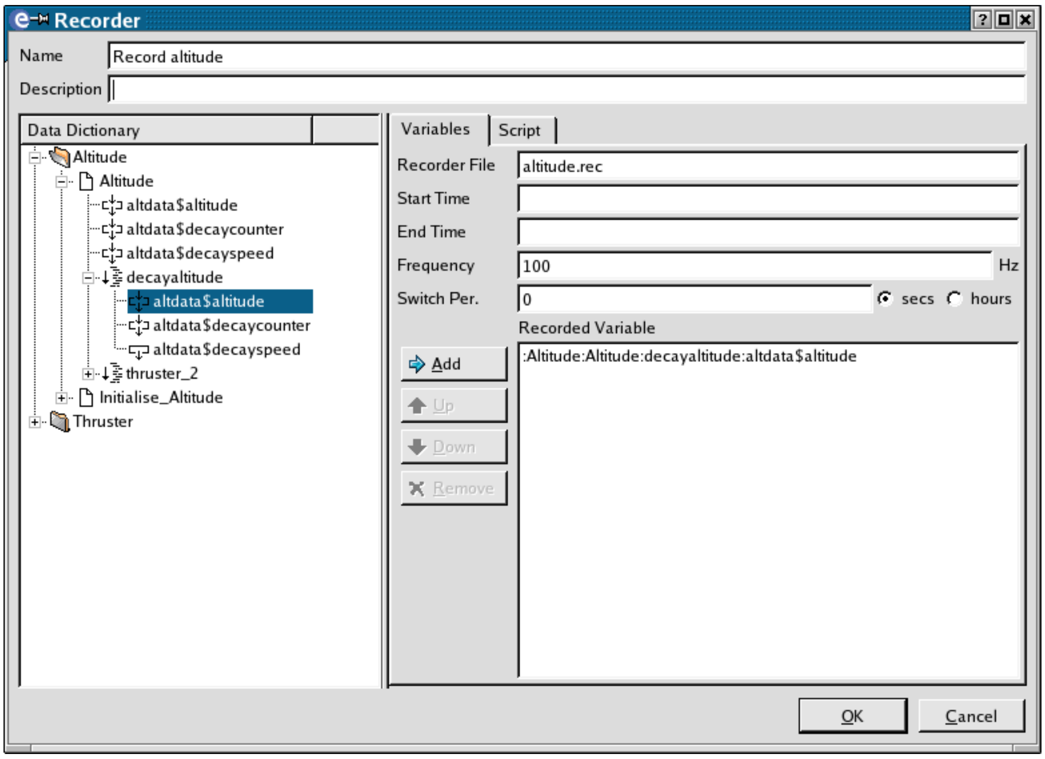
Figure 4.9: The Script Editor

4.8.3 Creating a recorder

In a recorder action, the values of one or more selected variables are saved to a file (in contrast with a monitor, where the values are shown on screen; another difference with monitors is the sample rate: monitors sample at a fixed rate of 2 Hz whereas recorders can sample at a user defined frequency up to the maximum schedule frequency, usually 200 Hz).

Select *Insert:New Recorder* to create a new recorder. In the Recorder Editor, change the name to `Record altitude`. Double click on the `altdata$altitude` variable in the Dictionary Browser. It will be added to the *Variables* list.

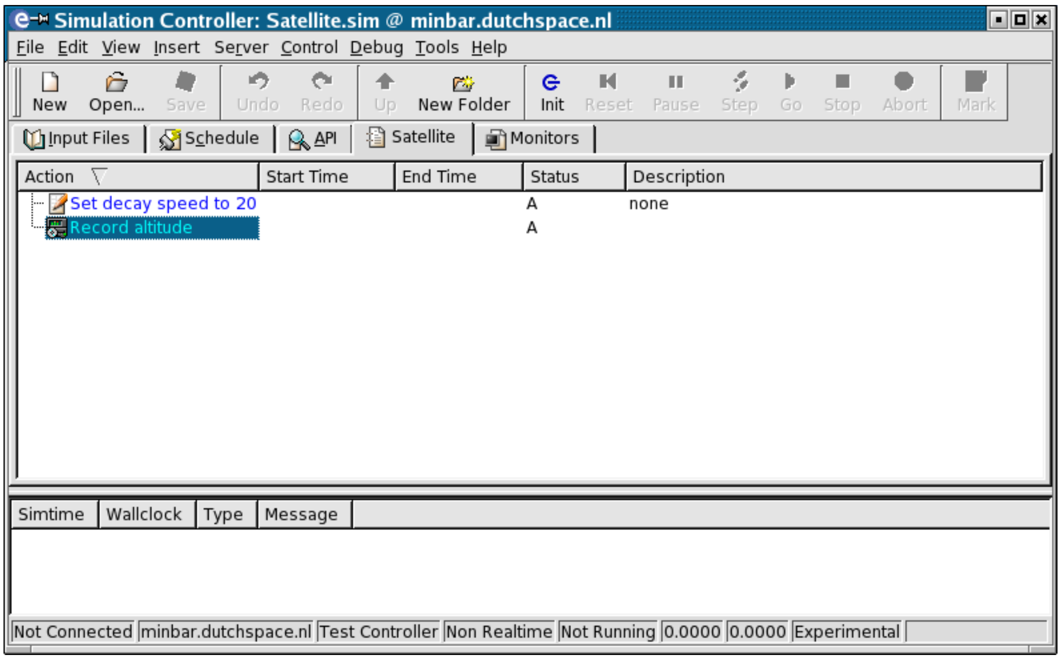
For a recorder, a number of extra attributes have to be filled in. Change the name of the recorder file by setting the edit field 'Recorder File' to `altitude.rec`. Optionally, the recording frequency and start/stop times can be entered here as well. The editor should now look like Figure 4.10.



Name: Record altitude
Description:
Recorder File: altitude.rec
Start Time:
End Time:
Frequency: 100 Hz
Switch Per.: 0 [✓ secs] hours
Recorded Variable
:Altitude:Altitude:decayaltitude:altdata$altitude

Figure 4.10: The Recorder Editor

The Recorder Editor has two tab pages. Change to the *Script* tab page, and notice that now a ‘Condition’ has been filled in: at a frequency of 100 Hz, the ‘Action’ will be executed. Although not used here, the ‘Inactive’ setting can be useful for temporarily disabling a recording action (or others, e.g. a check on variable values). Active actions are represented by an ‘A’ in the status column.
The Condition and Action fields are read only, but by checking the **Manual** checkbox you can customize these fields.
Close the Recorder Editor with the **OK** button. A second icon is now visible on the Scenario tab page. The tab page should now look like Figure 4.11.
Save the simulation definition by selecting *File:Save*. Requesting *Save* will cause the *Save As...* file selector to appear as this simulation definition has currently no filename. The simulation definition should be saved as `SUM.sim`.


Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API Satellite Monitors

Action	Start Time	End Time	Status	Description
Set decay speed to 20			A	none
Record altitude			A	

Simtime	Wallclock	Type	Message

Not Connected | minbar.dutchspace.nl | Test Controller | Non Realtime | Not Running | 0.0000 | 0.0000 | Experimental

Figure 4.11: The Scenario tab page

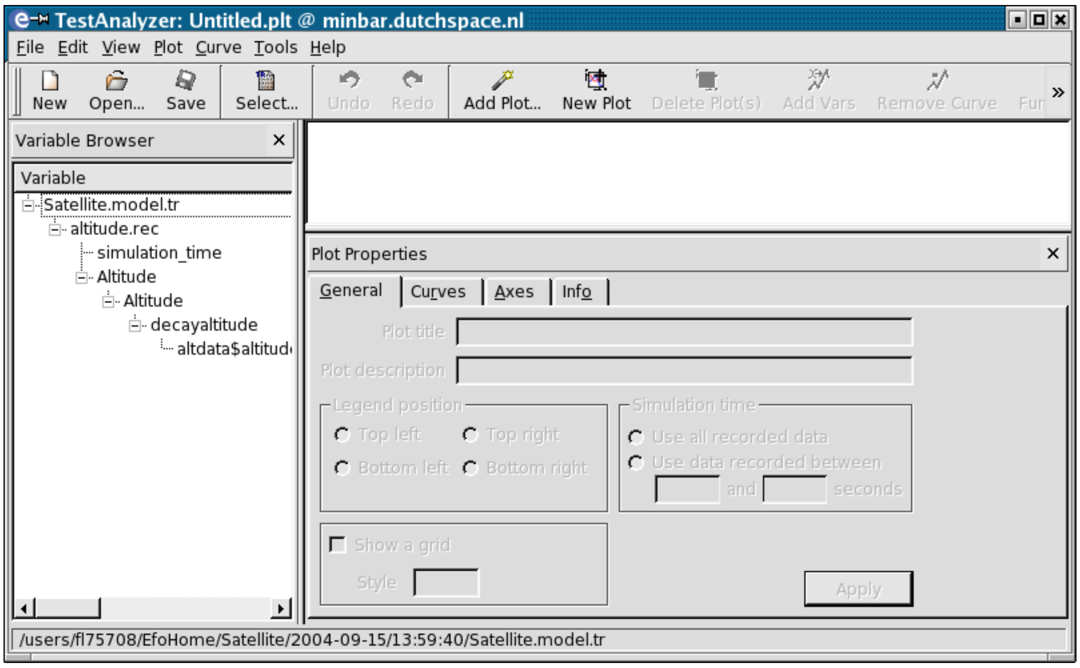
4.9 Executing a simulation run

Everything is now set to perform an actual simulation of the model. A simulation runs on a so-called *simulation server*, which is a machine running the EuroSim scheduler. Select *Server:Select server* from the menu, and select one of the servers shown in the list.
Simulations can run either in real time or non-real time. In non-real time mode, the simulation server will try to be as real time as possible, but no real-time errors will be generated (see also Section 2.2.5). By default, non-real time mode is selected.
Initialize the simulation by pressing the **Init** button from the tool bar or from the *Control:Init* menu. After the initialization is completed, the **Init** button will become inactive, and the other buttons will become active. Notice that the wall-clock time will start running.
Now press the **Go** button to start the simulation. On the Scenario tab page, notice that an ‘X’ appears in the status column for the recorder. This indicates that data is being recorded (the recorder is *eXecuting*). Select the Altitude tab page and notice that the altitude of the satellite is plotted against time in the monitor window. During the simulation, it is possible to change attributes of the monitor (for example the X and Y ranges).
When the satellite starts coming down, double-click on the ‘Set decay speed to 20’ intervention action. The satellite should now come down more rapidly. Directly after double clicking the intervention action, select *Insert:Mark Journal*. A mark with a number should now appear on the message pane. Afterwards, make a comment with *Insert:Comment Journal Mark* to explain that the mark indicates that the intervention action was executed. For example, enter as comment `Mark 1-tc indicates activation of intervention action.`
After a while, stop the simulation by pressing the **Pause** button and then the **Stop** button. Close the Simulation Controller with the *File:Exit* menu item.

4.10 Analyzing the simulation results

In order to make some plots of the recorded variables, select **Test Analyzer** from the main EuroSim window. Make sure you have PV-Wave or gnuplot installed otherwise this tool will not work. An empty Test Analyzer window will appear.

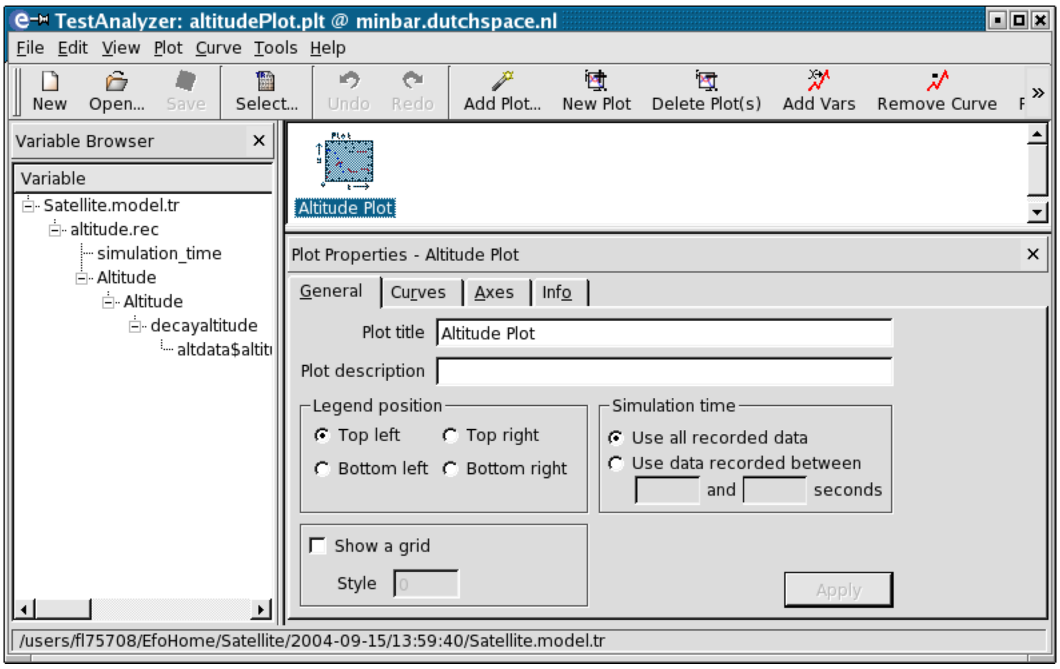
Now we will load the test results generated during the simulation. Select *File:Select Test Results File*. This will show a file selection window. Now find the recording file generated during the simulation. It will be in a directory like `2001-08-30/15:33:30`. Select the `Altitude.tr` file, which contains a list of all recording files created during the simulation (in this case, just one). Right click on the variable browser window (on the left) and select *Expand All Nodes*. The window should now look like Figure 4.12.



/users/fl75708/EfoHome/Satellite/2004-09-15/13:59:40/Satellite.model.tr

Figure 4.12: The Test Analyzer with the simulation results loaded

Now select *Plot:New Plot*. The plot view (top right) now shows an icon representing the plot. The plot properties tabpages (bottom right) have also become available.
Enter `Altitude` as the plot title and `Plot of altitude against time` as a description. Press the **Apply** button to commit the changes. The text under the plot icon in the plot view will be updated. The window should now look like Figure 4.13.



- Satellite.model.tr
  - altitude.rec
    - simulation_time
    - Altitude
      - Altitude
        - decayaltitude
          - altdata$altitu

Plot title: Altitude Plot

Plot description:

Legend position: [✓ Top left] Top right Bottom left Bottom right

Simulation time: [✓ Use all recorded data] Use data recorded between ___ and ___ seconds

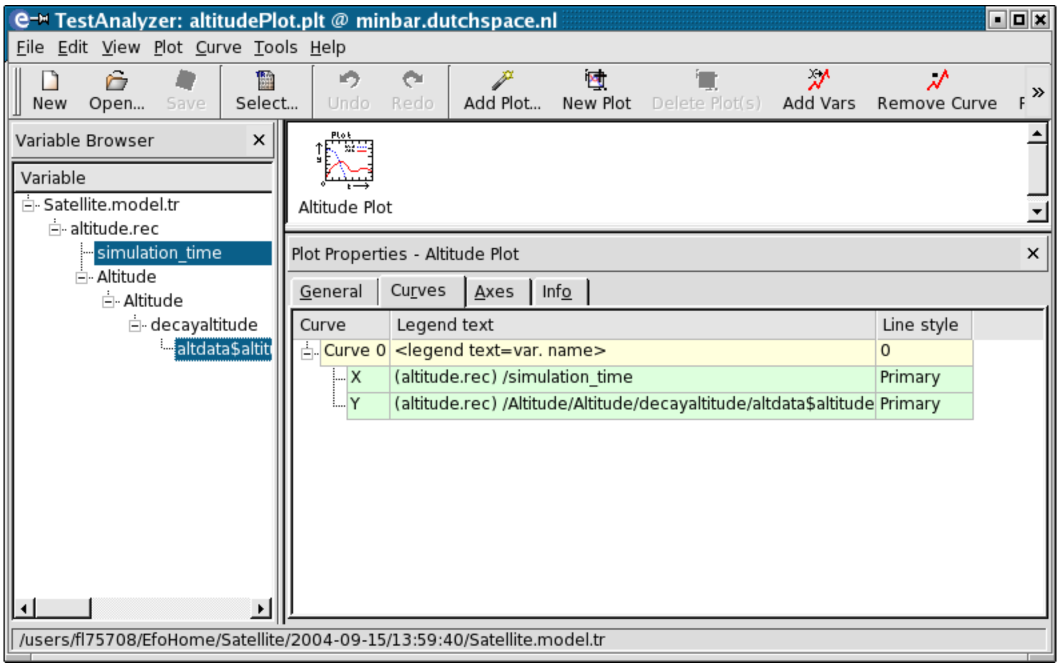
Show a grid [ ]

Style: 0

/users/fl75708/EfoHome/Satellite/2004-09-15/13:59:40/Satellite.model.tr

Figure 4.13: A new plot

The next step is to create a curve of the altitude versus the simulation time. Select the variable `altitude$altitude`. Now click on the variables and curves tab of the plot properties tabpages. The curve editor appears. Drag the selected variable from the variable browser to the curve editor. A new curve is created and the window should look like Figure 4.14.



- Satellite.model.tr
  - altitude.rec
    - simulation_time
    - Altitude
      - Altitude
        - decayaltitude
          - altdata$altit

Curve	Legend text	Line style
Curve 0	<legend text=var. name>	0
X	(altitude.rec) /simulation_time	Primary
Y	(altitude.rec) /Altitude/Altitude/decayaltitude/altdata$altitude	Primary

/users/fl75708/EfoHome/Satellite/2004-09-15/13:59:40/Satellite.model.tr

Figure 4.14: A completed plot

This completes the plot. Double clicking the plot icon in the plot view will show the plot.

4.11 Concluding remarks

In this chapter, a complete simulator has been built from scratch. The most important features of EuroSim have been used. However, as EuroSim offers many more functions than can be described in this tutorial, the reader is advised to proceed with the reference chapters, and experiment with the simulator from this chapter.

Part II

EuroSim Reference

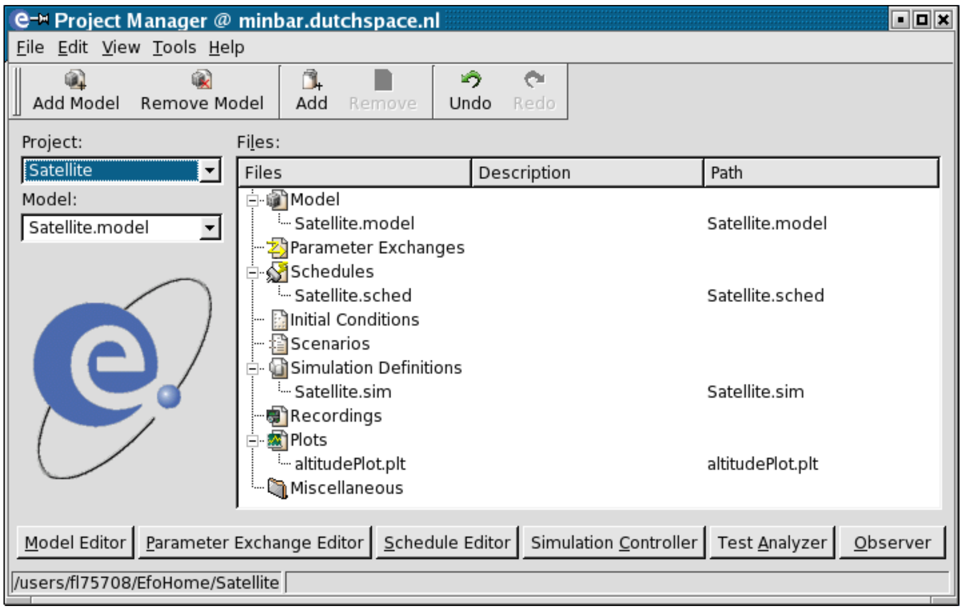
Chapter 5

EuroSim reference

This chapter describes the top-level interface of EuroSim (`esim`), the Project Manager. For a description of the various EuroSim components, such as the Model Editor and Schedule Editor, refer to the next chapters.

5.1 Starting EuroSim

The EuroSim environment is started with the esim command. This will pop-up the Project Manager window of EuroSim (see Figure 5.1).



Files	Description	Path
Model		
Satellite.model		Satellite.model
Parameter Exchanges		
Schedules		
Satellite.sched		Satellite.sched
Initial Conditions		
Scenarios		
Simulation Definitions		
Satellite.sim		Satellite.sim
Recordings		
Plots		
altitudePlot.plt		altitudePlot.plt
Miscellaneous		

/users/fl75708/EfoHome/Satellite

Figure 5.1: EuroSim start-up window

With the Project Manager the various editors can be started.
Before starting EuroSim, make sure that the environment variables `PATH`, `DISPLAY`[1], `EFOROOT` and `EFO_HOME` [2] are set correctly. On the RedHat Linux platform these environment variables are set automatically. On the IRIX and Windows NT platform environment variables are defined in the file `$EFOROOT/bin/esim.bashrc`. See also Section 4.2.
The Project Manager will use the global project database file `projects.db` in the directory pointed to by the `EFO_HOME` environment variable. If `EFO_HOME` has not been set before starting EuroSim, EuroSim will use the subdirectory `.eurosim` in your home directory.

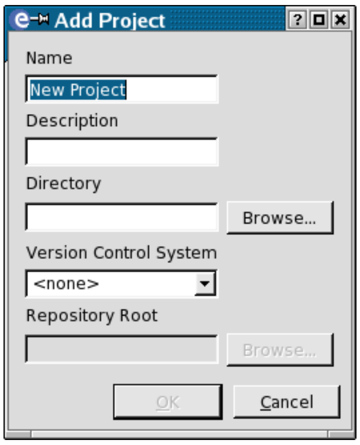
[1]On the Windows NT platform, the `DISPLAY` environment variable will not be used by EuroSim
[2]This variable only needs to be set to override the default value (`$HOME/.eurosim`)

The file `projects.db` contains all project references. If `projects.db` does not exist, EuroSim will create a new file.
Each individual project should have its own project directory (preferably in a subdirectory of `EFO_HOME`). This directory contains a local project database file `project.db`, which contains all references to project related models and files.
For a full list of the initialization files and environment files used by EuroSim, refer to Appendix F.
The Project Manager shows a screen with a **'Project'** combobox to select a project, a **'Model'** combobox to select a model and a **'Files'** list showing all files that refer to the model. Double clicking a file will start the associated editor. The editor buttons start an empty editor in the project directory.
EuroSim can be terminated by selecting the *File:Exit* menu option.

5.2 Menu items

5.2.1 File menu

Add Project...
: Opens a dialog for adding an existing project or for creating a new project (see Figure 5.2).



Name: New Project
Description:
Directory:
Version Control System: <none>
Repository Root:

Figure 5.2: Add Project dialog

Fill in the various project description items of the window. For the dialog field descriptions refer to Section 5.2.3, item "Project Settings...".

Remove Project
: Use this option to remove the current project from the projects list. The actual project files (such as the model file, the schedule, etc.) are not deleted.

5.2.2 Edit menu

Set Description...
: Adds a file description to a selected file.

Edit File...
: Opens the associated editor for the currently selected file. This is the same as double-clicking a file in the files list.

Add File...
: Opens a file dialog to add an existing file to the current model.

Remove File
: Removes the currently selected file from the model. The actual file is not deleted.

Add Model...
: Opens a file dialog to add an existing model file to the current project.

Remove Model
Removes the currently selected model file from the project. The actual model file and associated project files are not deleted.

5.2.3 Tools menu

Shell...
Opens a new command shell (e.g. xterm or a DOS command prompt).

Model Editor...
Starts the Model Editor.

Parameter Exchange Editor...
Starts the Parameter Exchange Editor.

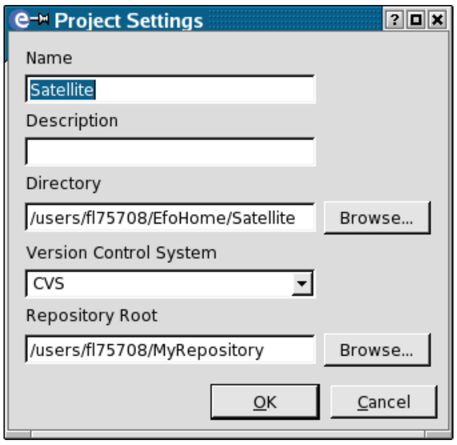
Schedule Editor...
Starts the Schedule Editor.

Simulation Controller...
Starts the Simulation Controller.

Test Analyzer...
Starts the Test Analyzer.

Observer...
Starts the Simulation Controller in Observer mode.

Project Settings...
Opens a dialog for changing various project description items. A project description contains a number of elements, each of which can be set in this dialog (see Figure 5.3).


Project Settings

Name: Satellite

Description:

Directory: /users/fl75708/EfoHome/Satellite

Version Control System: CVS

Repository Root: /users/fl75708/MyRepository

Figure 5.3: Project Settings dialog

Name The project name is the name that appears in the project list of the Project Manager, as well as in various other places, such as the name of the root node of the model hierarchy in the Model Editor.

Description
The project description is a free-text field that can be used for a more precise description of the project.

Directory
The project directory is the top of the directory tree in which all project related files will be stored. The **Browse** button can be used to search for an existing directory. Use the operating system file protections to protect project files against unauthorized use. Under UNIX one could for example create a UNIX group for each EuroSim project and

make the project files writable by group members only. Depending on the security level required, the project files can be made world readable or not[3].

Version Control System
: Defines which version control system will be used for this project. Currently EuroSim Mk3 supports the CVS and Cadese[4] version control systems.

Repository Root
: The repository root is the top of the directory tree in which the version management of the various model files will be stored. Refer to Section 2.6, for a discussion whether the repository can best be kept separate from the project root or not. The **Browse** button can be used to search for an existing directory. If an existing RCS or CVS repository is to be used within EuroSim, make sure that the tree under the project root has the same structure as the repository tree. The repository root field is optional and can be left empty. See Appendix M on how to set-up a repository root.

Preferences...
: Opens a dialog to set the preferences. The following items can be set.

Do not prompt to add files automatically
: When you start one of the EuroSim editors from the Project Manager and create a new file, you are prompted whether the new file should be added to the current project. If you check this item, you will not be prompted and the decision whether to add the file to the current project depends on the value of the next item.

Never add files automatically
: If this option is checked, new files that are created by one of the EuroSim editors will not be added to the current project automatically. If you want to add a newly created file afterward, then use the appropriate menu command.

5.2.4 Help menu

Online Help
: This menu option will start the 'Netscape' HTML-browser for UNIX and the 'Internet Explorer' for Windows NT which will load the on-line version of the user manual.

About EuroSim
: This will pop-up a window displaying the copyright information for EuroSim.

5.2.5 Automatic addition of files to the project

When you start one of the EuroSim editors from the Project Manager to create a new file (f.i. a new schedule file), the Project Manager will automatically add the new file to the current project when you save it to disk. Depending on the settings in the preferences dialog, you will be prompted with a question if the file should be added to the project or not. In the preferences dialog you can also disable this feature. Note that files other than model files, are always added in the context of the currently active model file in the current project. Each project can have multiple model files. If you have not yet selected a model file for the current project, the automatic addition of other files is disabled.

[3]Making UNIX groups and assigning members requires 'root' privileges and hence is a system administrators/facility managers job. Implementing a good protection strategy is not easy, but is assumed to be within the knowledge of the system administrator.

[4]Not supported in the Windows NT version.

Chapter 6

Model Editor reference

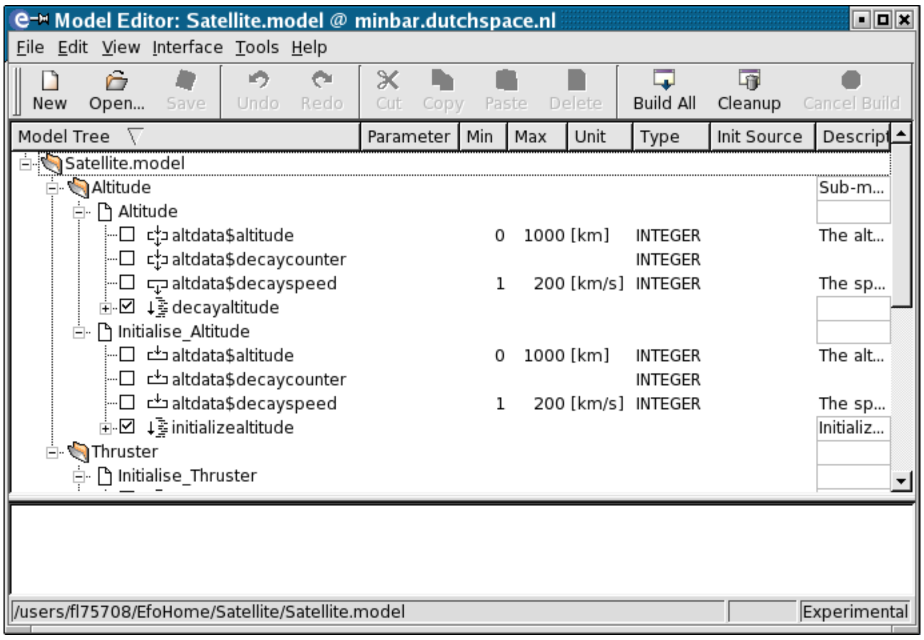
This chapter provides details on the Model Editor. The various objects which can be added to the model tree, the menu items of the editor and their options are described. For menu items not described in this chapter, refer to Section 3.5.

6.1 Starting the Model Editor

The Model Editor can be started by selecting the **Model Editor** button in the EuroSim start-up window (see Section 5.2.3).

6.2 Model Editor objects

In the Model Editor tree view the structure of the model is created using a hierarchical, tree structure. Elements in the tree are called *nodes* and have a specific function. The API (properties of variables and entrypoints available to the rest of EuroSim) can be edited in the Model Editor. In Figure 6.1 an example model tree is shown.



Model Tree	Parameter	Min	Max	Unit	Type	Init Source	Descript
Satellite.model							
Altitude							Sub-m...
Altitude							
☐ altdata$altitude		0	1000	[km]	INTEGER		The alt...
☐ altdata$decaycounter					INTEGER		
☐ altdata$decayspeed		1	200	[km/s]	INTEGER		The sp...
☑ decayaltitude							
Initialise_Altitude							
☐ altdata$altitude		0	1000	[km]	INTEGER		The alt...
☐ altdata$decaycounter					INTEGER		
☐ altdata$decayspeed		1	200	[km/s]	INTEGER		The sp...
☑ initializealtitude							Initializ...
Thruster							
Initialise_Thruster							

/users/fl75708/EfoHome/Satellite/Satellite.model Experimental

Figure 6.1: Example model tree

Note that only org nodes and file nodes can be directly added to the model hierarchy (using the menu options *Edit:Add Org Node*, *Edit:Add File Node*) or *Edit:Add Directory*. The other nodes are put into the

model hierarchy indirectly, e.g. by parsing the files. Informational messages are written to the logging window while parsing the files.
The next sections describe each of the nodes. The default icon for the node is shown in the left margin. If more than one icon is used, all are shown.

6.2.1 Root node

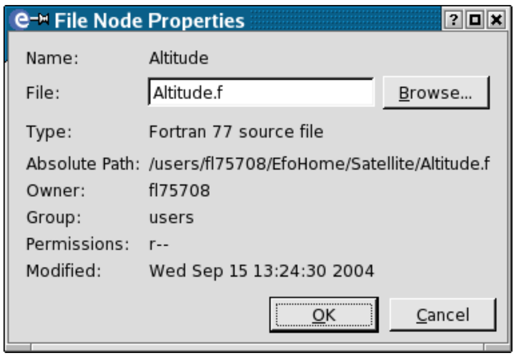
The root node represents the complete model. It is a special type of org node (see next section) and therefore shares the same attributes of org nodes. The name of the root node in the attributes window is the name of the model file. The name displayed on the Model Editor window is the (file)name of the model, or *Untitled.model* if a new model is started and has not been saved yet. Double-clicking the root node folds or unfolds the node.

6.2.2 Org node

Org nodes are used to structure the model. By using org nodes, two or more related sub-models can be grouped together by connecting them to the same org node. Both other org nodes as well as file nodes (representing the sub-models) can be attached to an org node.
The name of the org node can be changed by clicking a selected node. A description can be entered in the description field.

6.2.3 File node

There are various types of file nodes. They will be discussed in the sections below.
The name of the file node can be changed by clicking a selected node. The filename cannot be changed. A description can be entered in the description field.
The file attached to a file node can be viewed and edited through the menu options *Edit:View Source* and *Edit:Edit Source* respectively. Depending on the type of file, the correct viewer or editor is started. When a file is being edited or viewed the file icon with lock is shown.
The properties of a filenode can be shown with *Edit:Properties* (see Figure 6.2). You can select another file using the *Browse* button. For non-source files the type of the file can also be modified. As different file types have different attributes and functions, it is important to correctly enter the file type.


File Node Properties

Name:	Altitude
File:	Altitude.f
Type:	Fortran 77 source file
Absolute Path:	/users/fl75708/EfoHome/Satellite/Altitude.f
Owner:	fl75708
Group:	users
Permissions:	r--
Modified:	Wed Sep 15 13:24:30 2004

Figure 6.2: File Properties

See Section 3.5.4 for information on how to change the version requirement.

6.2.3.1 Source file node

Currently supported source file nodes are C files, FORTRAN files and Ada-95 files. For more information on the restrictions on those files, refer to Appendix H. Note that it is not possible to have more than one file node referring to the same source filename, even if these files are in different directories.

Double-clicking on a file node will unfold or fold it, thus showing the API information of a source file.
If the source file cannot be parsed, due to a syntax error, the broken file icon is shown. If the API information is changed, i.e. attributes of variables or entrypoints are changed, and the file is not yet saved the file icon gets an asterisk .
A variable or entrypoint is part of the API if its checkbox is checked. See the *decayaltitude* entry node in Figure 6.1.
Use the mouse or the **space** bar to change the state of the API check box on the current selection, which can contain multiple items.
Interface:Save API writes this information to the source file.
Warnings and errors that occur during parsing and saving of files are shown in the logging window at the bottom of the Model Editor.
For more information on how to add SMP source code and/or C++ sub-models, see Section 15.7.

6.2.3.2 FrameMaker file node

A FrameMaker file node can be used to attach documentation to a model. The file should be a FrameMaker MIF or regular document file. Only the MIF type of FrameMaker file can be versioned; the regular FrameMaker file will cause problems.

6.2.3.3 Environment file node

The environment node of a model is used to store information on the current development environment and the required target environment. It is used during build to check whether the current environment matches the required environment. The options *Edit:View Source* and *Edit:Edit Source* start the environment viewer and editor respectively. Refer to Section 6.6 for more information.

6.2.3.4 Model Description file node

Model Description file nodes can be added to the model file to generate a so called “datapool”. See Chapter 7 for a description on the datapool and how to create a Model Description file. During the build process (make), which can be started from the Model Editor, Model Description files that are part of the model will be read to generate the variables and entrypoints for the datapool.

6.2.4 Entry node

An entry node represents an entrypoint in a source file. It is part of the API of the model if its checkbox is checked (see Section 6.2.3.1).
The description is the only attribute of an entrypoint.
If the API information in the file contains entrypoints that are no longer available in the source code, a red cross is drawn through the icon.

6.2.5 Variable nodes

A variable node represents a variable in a source file. It is listed under the file where it is used and also under every entrypoint that uses it. It is part of the API of the model if its checkbox is checked. (See Section 6.2.3.1 above on API editing.)
The initial value and type of a variable are determined by parsing the source code.
Compound variables, such as arrays and structures, are shown as children of the variable node. Some attributes of the variable node can be edited at variable node level in the tree view of the Model Editor, while others must be edited at the variable base level (f.i. *min* and *max*). You can only edit attributes of variables when the API flag on the left of the variable is checked. Use the mouse or the **space** bar to change the state of the API check box on the current selection, which can contain multiple items.

A grey box around an attribute indicates that it is editable. Start editing by clicking in the box with the mouse or press the **F2** key to start editing the first editable attribute in the current selection. The **Tab** key moves to the next editable attribute in the current selection, while the **Enter** key finishes editing without moving to another attribute. The **Esc** key lets you leave edit mode without making any changes.
The user can specify:

- *parameter*: a variable set as a parameter may only be changed at initialization time by an initial condition.
- *unit*: the unit of the variable, e.g. *km*. It is for informational purposes only and written to the dictionary for use by other EuroSim tools, such as the API tab of the Simulation Controller.
- *min*: the minimum value of the variable.
- *max*: the maximum value of the variable.

The latter two (*min* and *max*) are checked at run-time when f.i. a user changes the value through the API tab of the Simulation Controller.
If the API information in the file contains variables that are not available in the source code a red cross is drawn through the icon.
Note that the entrypoint and variable information is extracted from the file after the language specific pre-processor has processed the file. In particular, if compile flags determine which entrypoints are available the API may show conflicts when compile flags change.
In order to avoid problems with globals that only have a local 'extern' declaration in entrypoints, the `extern` keyword will be emitted by EuroSim when creating the data dictionary. In particular this means that for externals with function scope no API information can be generated.

6.2.5.1 State variable

These nodes refer to variables which have filescope and are read and written by entrypoints in the file.

6.2.5.2 Input variable

These nodes refer to variables that are read by the entrypoint. The icon indicates that data is flowing out from the variable.

6.2.5.3 Output variable

These nodes refer to variables that are written by the entrypoint. The icon indicates that data is flowing into the variable.

6.2.5.4 Input/output variable

These nodes refer to variables that are both read and written by the entrypoint.

6.3 Selecting an API variable

6.3.1 Selection within a sub-model

When selecting a variable for inclusion within the API header, a variable can sometimes appear twice, because the parser sees the variable being used not only at file level, but also at the level of the function that uses it. See for example `altdata$altitude` in Figure 6.1.
In principle, there is no difference between selecting one or the other: both variable nodes are different representations of the same variable and hence point to the same memory address. The default situation can be taken as tagging variables at the level of their file scope. However, there can be sometimes reasons for tagging the variables beneath 'their' entrypoint:

- if there are a lot of API variables within a particular sub-model (source code file), then selecting variables which appear below their relevant entrypoints gives you an additional level of hierarchy which can ease identification and manipulation of API variables later on
- if there is a significant amount of data dependency between entrypoints which needs to be taken into account during scheduling, then again, the variables beneath entrypoints should be selected, as this relationship is used when determining tasks which share data (see also Section 11.3.5, on intersection)

6.3.2 Selection from two or more sub-models

Where variables are used by two or more functions, they will appear in more than one sub-model. An example is the `altdata$altitude` variable seen in Figure 6.1, which also appears in the listing of variables for the `Initialise_Altitude` source file.
Again, there is no difference between selecting one or the other, as both representations point to the same memory address. The general guideline is to tag (and annotate) the variable belonging to the code which will be active during the executing scheduling state. In the example given above, this means that `altdata$altitude` would be tagged for the `Altitude` source rather than for its one-off use in the `Initialise_Altitude` source.

6.4 Menu items

6.4.1 File menu

New Creates a new empty model.

Open Opens a model.

Save Save the current model.

Save As If the model file is saved to a different directory, the file nodes are updated so that the newly saved model file shares its files with the original model file. If you want a copy of the model file with the relative pathnames of file nodes unchanged, thus possibly referring to non-existing files, use the UNIX `cp` or DOS copy command from the command line of a shell.

Exit Exit the Model Editor.

6.4.2 Edit menu

Undo/Redo
Undo/redo actions.

Cut/Copy
When cutting or copying an org node, the whole subtree, including the selected org node, will be copied for later pasting.

Paste Paste cut or copied data. Nodes are pasted into the currently selected node.

Delete Delete the current selection.

Add Org Node...
When an org-node is selected in the model hierarchy, this menu item can used to attach a new org node as a child to the selected node. The name and description of the new node can be entered.

Add File Node...
When an org-node is selected in the model hierarchy, this menu item can be used to attach a new file node as a child to the selected node. The file can be selected using a file selector. The name of the node can be changed into a more descriptive name by clicking in the selected node

name after the file node has been added to the node tree. When adding a non-existing file, a dialog box will pop-up asking whether to create a new file or not. Templates for new files can be found in the `lib/templates` sub-directory of the EuroSim installation directory.

Add Directory. . .
: When an org-node is selected in the model hierarchy, this menu item can be used to recursively add a complete directory tree to the selected node. The directory can be selected using a directory selector. Each directory found in the selected directory will be added as an org-node. The files that are found will be added as children to their respective parent node. This command automatically filters out the `CVS` directories, if any.

Edit Source
: For file nodes, this option will start an editor with which the file attached to the node can be modified. For program source files by default the 'vi' editor will be started on UNIX platforms and NotePad on Windows NT platforms. If the environment variable `EDITOR` is set, that editor will be used.

For environment file nodes, the environment editor (see Section 6.6) will be started.

View Source
: For file nodes, this option will start (if applicable) an external program to view the contents of the file attached to the node.

Find Node
: With the Find Node option, it is possible to search through the model hierarchy for a certain node. (see Figure 6.3).

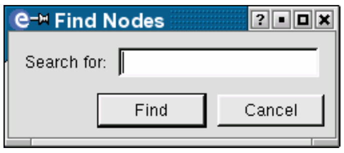


Figure 6.3: Search window

Rename Node
: Rename the currently selected file or org node.

Properties
: Shows the properties of a file node (see Figure 6.2) and allows specifying another file name for this file node.

Clear Min
: Clears the minimum value(s) of a variable node.

Clear Max
: Clears the maximum value(s) of a variable node.

6.4.3 View menu

Expand To Files
: This menu option will show file nodes.

Expand All
: This menu option will show all nodes of the tree. All source files will be parsed and entrypoints and variables will be shown.

Collapse All
: This menu option will close all nodes of the tree.

6.4.4 Interface menu

Include Add variable or entrypoint to the API.

Exclude
Remove a variable or entrypoint from the API

Exclude all undefined. . .
Remove all variables and/or entrypoints that are still in the API but no longer available in the sub-model source code.

Parse File(s)
Parse the selected file(s) to discover it's API and/or find items that can be added to the API of the sub-model.

Save API
Writes the API information to the sub-model source file.

Clear API
Removes the API information from the sub-model source file.

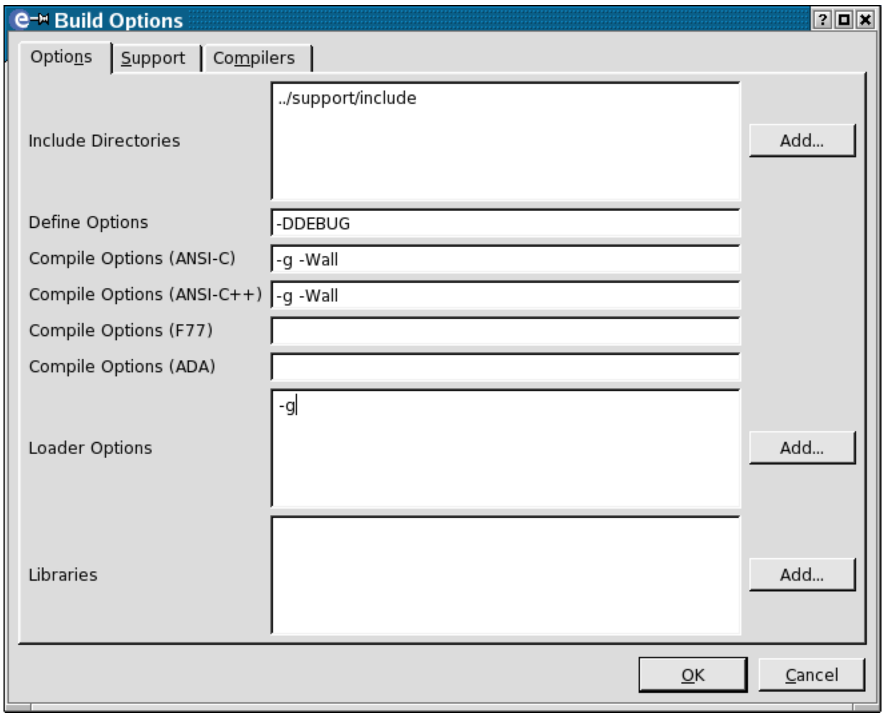
6.4.5 Tools menu

Build All
Build the simulator and data dictionary.

Build Clean
This menu option will remove all generated files from the model directory. This includes the data dictionary, and compiler generated object files. Use this option to force a rebuild of the model. This option is generally used when a new version of EuroSim has been installed, when the filesystem has had integrity problems, or when EuroSim does not behave as expected.

One specific case where a clean up is required is when you add a new file to the model hierarchy (e.g. a C source file) which is older than the already existing target file (e.g. add a file `file.c` whilst there still is a newer `file.o`). The `make` which is used to build the simulator will then not know that the target should be recreated. The same applies when deleting a file node from the model tree.

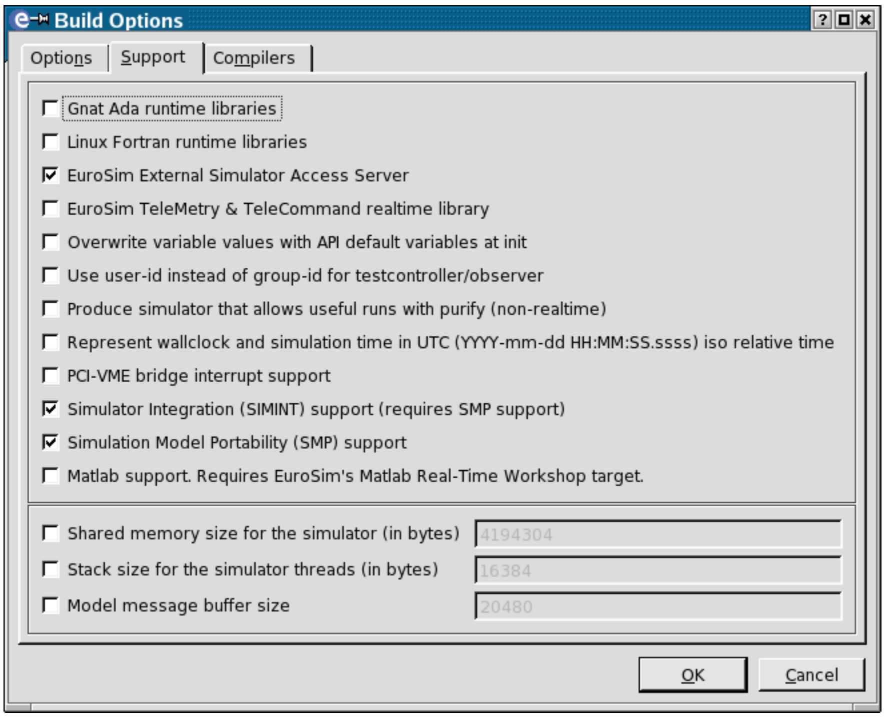
Set Build Options. . .
When in source files external functions are used (such as arithmetic or string functions), the libraries containing these functions can be specified in the options dialog shown by this menu option (see Figure 6.4).



Include Directories	../support/include
Define Options	-DDEBUG
Compile Options (ANSI-C)	-g -Wall
Compile Options (ANSI-C++)	-g -Wall
Compile Options (F77)	
Compile Options (ADA)	
Loader Options	-g
Libraries	

Figure 6.4: Model Build Options dialog: Options tab page

Also, specific compiler options can be specified, including directories where the compilers should look for include files. In the libraries field, libraries which need to be linked to the simulator should specified in the form `-l`*libraryname*. One of the more often used libraries is 'm', the math library.



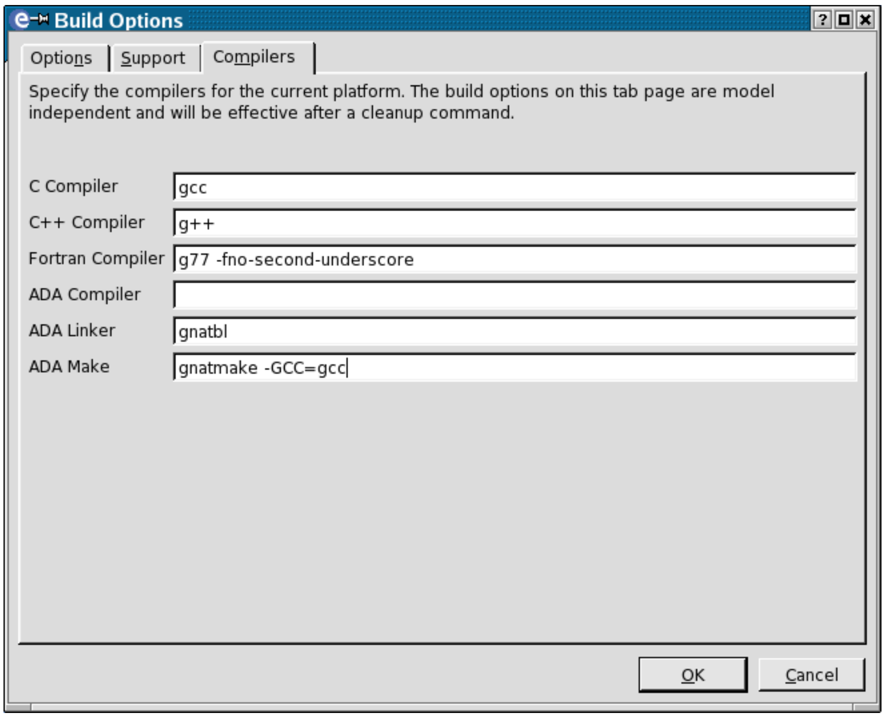
- ☐ Gnat Ada runtime libraries
- ☐ Linux Fortran runtime libraries
- ☑ EuroSim External Simulator Access Server
- ☐ EuroSim TeleMetry & TeleCommand realtime library
- ☐ Overwrite variable values with API default variables at init
- ☐ Use user-id instead of group-id for testcontroller/observer
- ☐ Produce simulator that allows useful runs with purify (non-realtime)
- ☐ Represent wallclock and simulation time in UTC (YYYY-mm-dd HH:MM:SS.ssss) iso relative time
- ☐ PCI-VME bridge interrupt support
- ☑ Simulator Integration (SIMINT) support (requires SMP support)
- ☑ Simulation Model Portability (SMP) support
- ☐ Matlab support. Requires EuroSim's Matlab Real-Time Workshop target.

☐ Shared memory size for the simulator (in bytes)	4194304
☐ Stack size for the simulator threads (in bytes)	16384
☐ Model message buffer size	20480

Figure 6.5: Model Build Options dialog: Support tab page

Figure 6.5 shows the available pre-defined build support options for the simulator. Selecting one or more of these options causes libraries such as 'external simulator' or 'telemetry and telecommand' to be linked in, augmenting the simulator with extra runtime functions. Usage of

Ada-95 and/or Fortran runtime libraries requires explicit selection of the appropriate options. Options are described in the EuroSim.capabilities manual page, and can be listed using the `esimcapability` command.



Specify the compilers for the current platform. The build options on this tab page are model independent and will be effective after a cleanup command.

C Compiler	gcc
C++ Compiler	g++
Fortran Compiler	g77 -fno-second-underscore
ADA Compiler	
ADA Linker	gnatbl
ADA Make	gnatmake -GCC=gcc

Figure 6.6: Model Build Options dialog: Compilers tab page

The Compilers tab page (see Figure 6.6) allows you to specify which *compiler(s)* and related utilities to use to build the simulator. When specifying a command, the default used by the build command will be overruled. Leaving a field blank in the dialog will cause the build command to use the default command.

You can specify just the command (provided its directory can be found in the `PATH` environment variable) or the full path, for example:

```
/usr/bin/gcc
```

You can also specify additional command line options for a specific command, for example:

```
g77 --no-second-underscore
```

The commands specified on this tab page dialog are not stored in the model file, but in a global resource[1]. Therefore, the command specifications are model independent. The specifications are read by the ModelMake utility when generating the makefile that is used to build the simulator executable. They are effective after the *Tools:Cleanup* command.

Clear Logging
Clears the logging window at the bottom of the Model Editor.

Save Logging
Opens a file dialog where you can select or specify the name of the file to save the contents of the logging window.

Preferences
Shows a dialog where you can specify your preferences, such as always saving API information to files and saving the changes to the .model file or automatically clearing the logging window, before starting a build.

[1]Located in the `.eurosim` sub-dicrectory of your home directory (Unix systems) or in the registry (Windows systems)

6.5 Model Editor Preferences

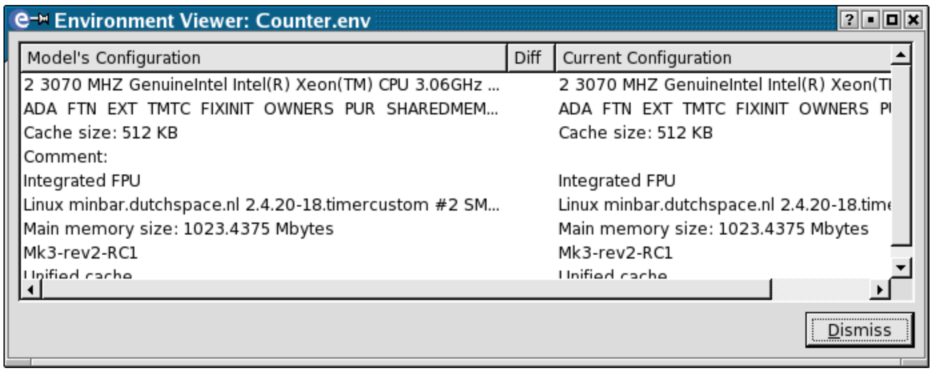
The Model Editor can be customized for a number of aspects (e.g. editors to start for different types of node). The system wide preferences can be found in the `$EFOROOT/etc/esim_conf` file. To make personal customizations, one can overrule any setting with an `esim_conf` file in your home directory. Model Editor specific preferences and preferences related to version control can be changed using the Preferences dialog (see menu *Tools:Preferences*).

6.6 The environment editor and viewer

The environment editor is started by selecting the environment node in the model tree and selecting the *Node:Edit Source* menu option. The viewer is started using the menu option *Node:View Source* when the environment node is selected.

6.6.1 The environment viewer

The environment contains information on the target hardware required for the simulator being developed. The environment viewer (see Figure 6.7) shows at the right the current environment, and at the left the target environment, as it is stored in the environment file. If there are any differences between the two, these are indicated with unequal signs (<>).
If a field from the environment is too long to fit in the text area, the middle mouse button can be used to scroll the text area to reveal the remainder of the field.

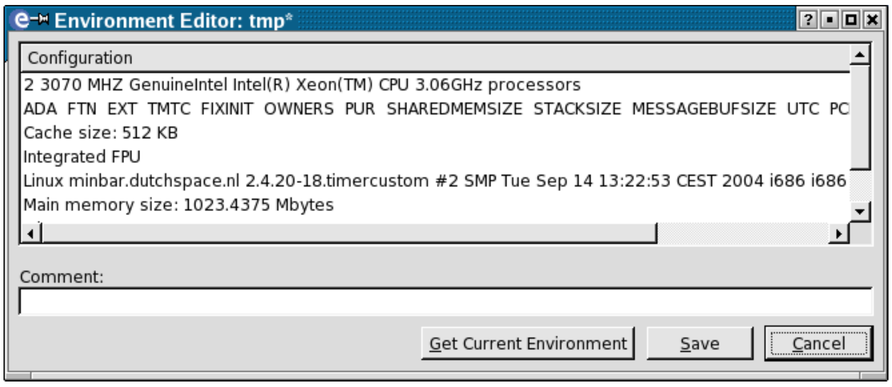


Model's Configuration	Diff	Current Configuration
2 3070 MHZ GenuineIntel Intel(R) Xeon(TM) CPU 3.06GHz ...		2 3070 MHZ GenuineIntel Intel(R) Xeon(Tl
ADA FTN EXT TMTC FIXINIT OWNERS PUR SHAREDMEM...		ADA FTN EXT TMTC FIXINIT OWNERS PI
Cache size: 512 KB		Cache size: 512 KB
Comment:		
Integrated FPU		Integrated FPU
Linux minbar.dutchspace.nl 2.4.20-18.timercustom #2 SM...		Linux minbar.dutchspace.nl 2.4.20-18.tim
Main memory size: 1023.4375 Mbytes		Main memory size: 1023.4375 Mbytes
Mk3-rev2-RC1		Mk3-rev2-RC1
Unified cache		Unified cache

Figure 6.7: The environment viewer

6.6.2 The environment editor

The environment editor allows the user to retrieve the current environment and save it to the environment description file, as well as adding a comment to the environment file.
Use the button *Get Current Environment* in the Environment Editor to retrieve the current environment.
To put the file under configuration control use the same procedure as for source code files.



```
2 3070 MHZ GenuineIntel Intel(R) Xeon(TM) CPU 3.06GHz processors
ADA  FTN  EXT  TMTC  FIXINIT  OWNERS  PUR  SHAREDMEMSIZE  STACKSIZE  MESSAGEBUFSIZE  UTC  PC
Cache size: 512 KB
Integrated FPU
Linux minbar.dutchspace.nl 2.4.20-18.timercustom #2 SMP Tue Sep 14 13:22:53 CEST 2004 i686 i686
Main memory size: 1023.4375 Mbytes
```

Figure 6.8: The environment editor

Chapter 7

Model Description Editor reference

This chapter provides details on the Model Description Editor (MDE). The menu items that are specific to the MDE will be described in separate subsections of this chapter. For menu items not described in this chapter, refer to Section 3.5.

7.1 Introduction

The use of the MDE is optional, but you would typically use Model Description files when integrating several independent models into one simulator without wanting to do the integration explicitly in (model) source code. Use Model Description files in combination with Parameter Exchange files (see Chapter 8) to exchange data between models.

Model Description files serve as input to functions of the Simulator Integration Support library, which is described in detail in Chapter 19.

The MDE can be used to create one or more Model Description files that describe copies of API variables[1] that exist in the data dictionary. The data dictionary itself is built by the build process (make) that can be started from the EuroSim Model Editor, see Section 6.4.5.
The copies of the variables can have names that are different from the ones in the data dictionary. This is especially useful when the data dictionary contains API variables with ambiguous names (f.i. when the source code of the model is generated by a software generation tool) or when you address an index in an array variable and wish to give it a more descriptive name, for example:

```
model description      data dictionary

sun/update/input/X     sun.c/vector[0]
sun/update/input/Y     sun.c/vector[1]
sun/update/input/Z     sun.c/vector[2]
```

7.2 Datapool

All variables created by the MDE (i.e. the copies of the API variables) will be added to a special node in the data dictionary, the so called "datapool". In order to update these variables in the datapool, special entrypoints are automatically generated. These entrypoints contain the source code to copy the values of the variables of the model to the copies in the datapool (in case of output variables) or vice versa (in case of input variables). The datapool and the generated entrypoints are merged into the data dictionary during the last step of the build process so that the datapool variables and entrypoints are available to the EuroSim simulator.

[1]An API variable is a model variable that is marked in the Model Editor to be exported to the data dictionary.

7.2.1 User defined variables

The MDE supports creation of user defined variables in the datapool. User defined variables are variables that do not have a relation with a model API variable. Typical use of user defined datapool variables is with EuroSim External Simulator Access, see Chapter 18. The user defined variables in the datapool are f.i. updated by an external client.

7.3 Scheduling datapool updates

The automatically generated entrypoints must be called by the scheduler at the appropriate time steps, see Figure 7.1 for a very simple example of a datapool and model source code. At step 1 the automatically generated entrypoint takes care of copying the value of the X variable in the datapool to the X variable of the model code. Step 2 calls the actual entrypoint in the model to update the X variable. At last, step 3 copies the updated model variable X back to the datapool. This last step is also performed by automatically generated code. Use the Schedule Editor to specify when the generated entrypoints should be called. The generated entrypoints are also placed under the datapool node in the data dictionary. The names of the entrypoints are based on the names of the input and output group nodes.

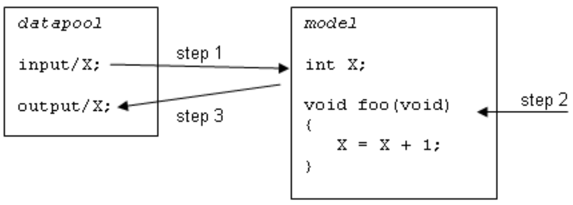


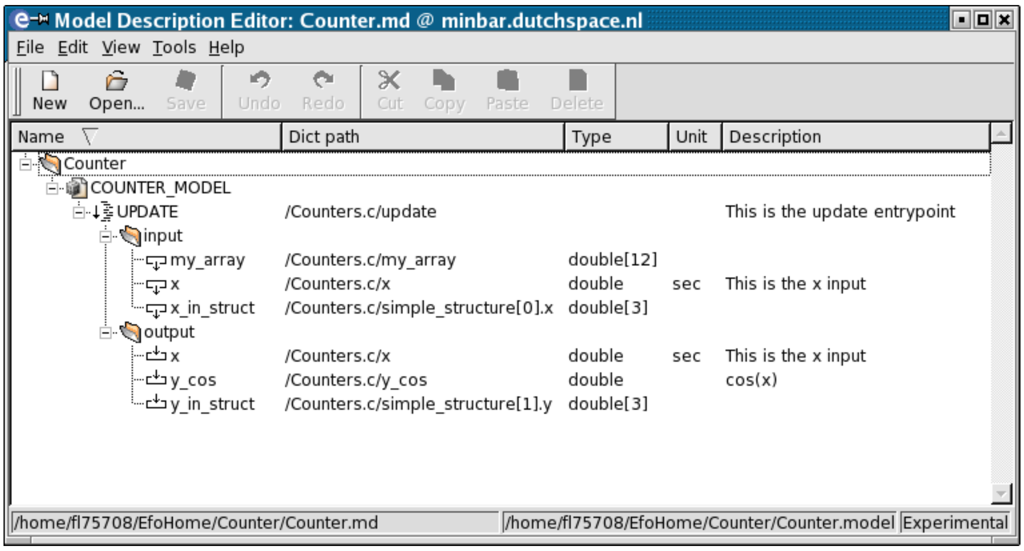
Figure 7.1: Example of data transfer between datapool and model

7.4 Starting the Model Description Editor

The Model Description Editor (MDE) can be started from the Model Editor. When the model tree contains a file with the appropriate extension (see Appendix F), then the MDE is automatically started when the Edit command is selected on the model description file node in the Model Editor.
The MDE needs a data dictionary as input. When the MDE is started from the Model Editor, the Model Editor first runs the build process (make) in order to ensure that the data dictionary is up to date. This means that there may be some delay when starting the MDE if there are a lot of outstanding changes since the last build command was given.

7.5 Model Description Editor objects

In the Model Description Editor tree view the model description is created using a hierarchical tree structure. Elements in the tree are called *nodes* and have a specific function.
In Figure 7.2 an example model description tree is shown.


Model Description Editor: Counter.md @ minbar.dutchspace.nl
File Edit View Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete

Name	Dict path	Type	Unit	Description
Counter				
COUNTER_MODEL				
UPDATE	/Counters.c/update			This is the update entrypoint
input				
my_array	/Counters.c/my_array	double[12]		
x	/Counters.c/x	double	sec	This is the x input
x_in_struct	/Counters.c/simple_structure[0].x	double[3]		
output				
x	/Counters.c/x	double	sec	This is the x input
y_cos	/Counters.c/y_cos	double		cos(x)
y_in_struct	/Counters.c/simple_structure[1].y	double[3]		

/home/fl75708/EfoHome/Counter/Counter.md /home/fl75708/EfoHome/Counter/Counter.model Experimental

Figure 7.2: Example model description tree

7.5.1 Root node

Each model description has one root node. It represents the complete model description and it has the basename of the model description file. The root node can hold one or more Model nodes.

7.5.2 Model node

Model nodes are used to structure the model description and will usually (but not necessarily) refer to the model(s) as specified in the Model Editor. Model nodes are children of the root node and can hold one or more Entrypoint nodes.

7.5.3 Entrypoint node

Entrypoint nodes are also used to structure the model description and refer to an entrypoint in the model code. Entrypoint nodes are children of a model node and can hold inputs and outputs group nodes.
When you create a new Entrypoint node, you are presented with a dialog box to select an entrypoint from the data dictionary.

7.5.4 Inputs and Outputs group nodes

Inputs and Outputs group nodes are used to logically group the input and output variables of an entrypoint. Inputs and Outputs group nodes are children of an Entrypoint node. An Inputs group node can hold Input nodes and an Outputs group node can hold output nodes.

7.5.5 Input and Output nodes

Input and Output nodes refer to API variables of the model code (i.e. variables in the data dictionary) or they are user defined (i.e. the node holds an ANSI-C variable declaration). Input and Output nodes cannot have children, i.e. they are the leaves of the model description tree.
When you create a new input or output node, you are presented with a dialog box to select the API variable from the data dictionary or enter an ANSI-C variable declaration when defining a user defined variable. In the latter case, the name of the node is derived from the entered variable name.

7.6 Menu items

Note that most common commands are also available in context sensitive menus that pop-up when clicking the right mouse button. Some commands also have keyboard short-cuts.

7.6.1 File menu

Select model
Select the model file that will be used to get the data dictionary. The model file (and hence the data dictionary) defines which entrypoints and variables you can choose from in the dialogs when adding and entrypoint node or a variable node.

7.6.2 Edit menu

Add Model node
Add a model node to the root node, see Section 7.5.2.

Add Entrypoint node
Add an Entrypoint node to a Model node, see Section 7.5.3.

Add Inputs group node
Add an Inputs group node to an Entrypoint node, see Section 7.5.4.

Add Outputs group node
Add an Outputs group node to an Entrypoint node, see Section 7.5.4.

Add Input node
Add an Input node to an Inputs group node, see Section 7.5.5.

Add Output node
Add an Output node to an Outputs group node, see Section 7.5.5

7.6.3 Tools menu

Check Model Description for errors
Checks the model description for any errors. The model description is also automatically checked on each save to disk. This feature can be disabled through the Tools:Preferences menu.

Chapter 8

Parameter Exchange Editor reference

This chapter provides details on the Parameter Exchange Editor (PXE). The menu items that are specific to the PXE will be described in separate subsections. For menu items not described in this chapter, refer to Section 3.5.

8.1 Introduction

The use of the PXE is optional, but you would typically use Parameter Exchange files when integrating several independent models into one simulator without wanting to do the integration explicitly in (model) source code. Use Parameter Exchange files in combination with Model Description files (see Chapter 7) to exchange data between models.

Parameter Exchange files serve as input to functions of the Simulator Integration Support library, which is described in detail in Chapter 19.

The PXE can be used to create one or more Parameter Exchange files that describe which output variables in the datapool should be copied to which input variables in the datapool, see Section 7.2 for a brief description on how to create the datapool using the EuroSim Model Description Editor.

8.2 Scheduling parameter exchanges

The actual copy of the variables is performed by automatically generated entrypoints. These entrypoints are placed in a special node of the data dictionary, called "paramexchg". The entrypoints have the same name as the exchange group. Exchange groups are described later on in this chapter. There is no need to re-build the data dictionary in the EuroSim Model Editor, since the entrypoints are generated at run-time by reading the appropriate Parameter Exchange files. Use the *File* menu in the EuroSim Schedule Editor to specify which Parameter Exchange files should be used for the simulator, see Section 11.3.1.
A simple example of scheduling an exchange group entrypoint is given in Figure 8.1. After model A has been updated and its output variable in the datapool is set (see Scheduling datapool updates in Section 7.3), the parameter exchange can take place between model A and model B. This also shows that scheduling the exchange has to be done at the appropriate point in time, i.e. *after* all models have updated their output variables and *before* the (other) models need the updated data on their respective input variables.

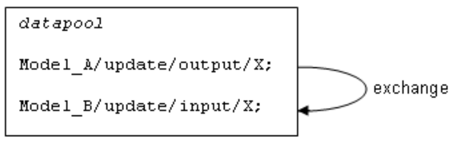


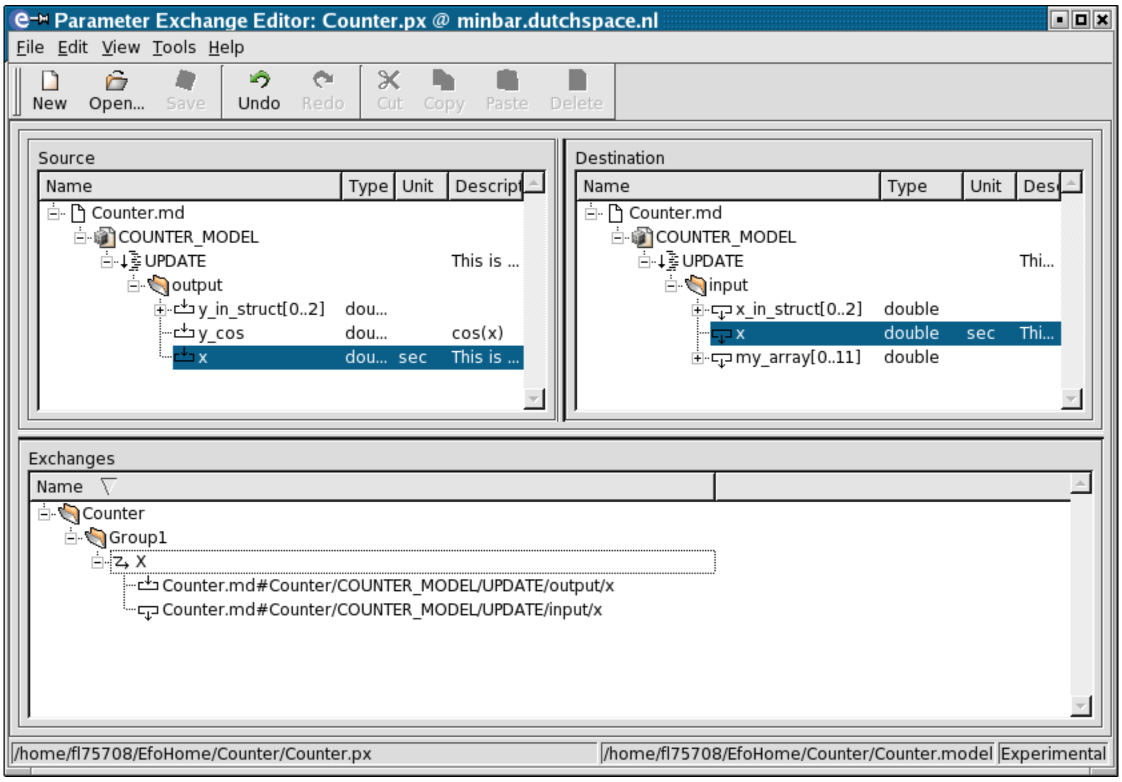
Figure 8.1: Example of data transfer between models

8.3 Starting the Parameter Exchange Editor

The Parameter Exchange Editor (PXE) can be started by selecting the **Parameter Exchange Editor** button in the EuroSim start-up window (see Section 5.2.3).
The PXE uses the data dictionary and one or more Model Description files as input.

8.4 Parameter Exchange Editor objects

The Parameter Exchange Editor has three views. The source and destination views are read-only and show the available output and input variables of the selected Model Description files. Use the *File* menu to add Model Description files to the views. The exchange view provides a view of the parameter exchanges and can be used to add, remove, update and rename the actual parameter exchanges.
In Figure 8.2 an example parameter exchange tree is shown in the bottom view.


Parameter Exchange Editor: Counter.px @ minbar.dutchspace.nl
File Edit View Tools Help
New
Open...
Save
Undo
Redo
Cut
Copy
Paste
Delete
Source
Destination
Exchanges

Source

Name	Type	Unit	Descript
Counter.md			
COUNTER_MODEL			
UPDATE			This is ...
output			
y_in_struct[0..2]	dou...		
y_cos	dou...		cos(x)
x	dou...	sec	This is ...

Destination

Name	Type	Unit	Des
Counter.md			
COUNTER_MODEL			
UPDATE			Thi...
input			
x_in_struct[0..2]	double		
x	double	sec	Thi...
my_array[0..11]	double		

Exchanges

```
Name
Counter
  Group1
    x
      Counter.md#Counter/COUNTER_MODEL/UPDATE/output/x
      Counter.md#Counter/COUNTER_MODEL/UPDATE/input/x
```

/home/fl75708/EfoHome/Counter/Counter.px | /home/fl75708/EfoHome/Counter/Counter.model | Experimental

Figure 8.2: Example parameter exchange tree

8.4.1 Source view

The source view provides an overview of available output variables in the selected Model Description files. Use the *File* menu to add Model Description files to the view.

8.4.2 Destination view

The destination view provides an overview of available input variables in the selected Model Description files. Use the *File* menu to add Model Description files to the view.

8.4.3 Exchange view

The exchange view is used to view and edit the parameter exchange by means of a hierarchical tree structure. Elements in the tree are called *nodes* and have a specific function, which are described in the following subsections.

8.4.3.1 Root node

Each parameter exchange has one root node. It represents the complete parameter exchange and it has the basename of the parameter exchange file. The root node can hold one ore more Exchange group nodes.

8.4.3.2 Exchange group node

An exchange group is used to organize a logical group of exchanges for which the exchange (copy) of variables can be scheduled as one step. For each exchange group an entrypoint will be generated with the same name as the exchange group under the “paramexchg” node in the data dictionary. An exchange group node contains the actual exchange parameters.

8.4.3.3 Exchange parameter node

An exchange parameter specifies which output variable from the datapool - as specified by a Model Description file - should be copied to which input variable in the datapool. The value of the output variable is copied to the specified input variable by an automatically generated entrypoint that has the name of the parent exchange group node. You must specify when to schedule this entrypoint using the EuroSim Schedule Editor. An exchange parameter is a child of an exchange group node and it cannot have children, i.e. it is the leaf of the parameter exchange tree.

8.5 Menu items

Note that most common commands are also available in context sensitive menus that pop-up when clicking the right mouse button. Some commands also have keyboard short-cuts.

8.5.1 File menu

Add Model Description
Add a Model Description file to the source and destination views.

Select model
Select the model file that will be used to get the data dictionary. The data dictionary is used to check if a parameter exchange is valid, i.e. it checks the type and size of the source and destination variable.

8.5.2 Edit menu

Add Exchange Group
Add an exchange group node to the root node, see Section 8.4.3.2.

Add Exchange Parameter
Add an exchange parameter to an exchange group node, see Section 8.4.3.3. You will be prompted with a dialog box to enter a name (a sensible default is provided). The name is purely informational. In order to add an exchange parameter you must first select an output variable in the source view and an input variable in the destination view. Then select the appropriate exchange group and select the *Add Exchange Parameter* command in the Edit menu.

Update Exchange Parameter
Update an exchange parameter with currently selected input and output variables in the destination and source views, respectively.

8.5.3 Tools menu

Check Parameter Exchange for errors
Checks the parameter exchange for any errors. The parameter exchange is also automatically checked on each save to disk. This feature can be disabled through the Tools:Preferences menu.

Check Coverage
Check if all output and input variables are covered by exchanges.

Chapter 9

Calibration Editor reference

This chapter provides details on the Calibration Editor (CE). The menu items that are specific to the CE will be described in separate subsections. For menu items not described in this chapter, refer to Section 3.5.

9.1 Introduction

The use of the CE is optional, but you would typically use Calibration files when you need to interface with external hardware such as electrical front-ends.

Calibration files serve as input to functions of the Calibration library, which is described in detail in Section 9.6.1.

The CE can be used to create one or more Calibration files that describe the transformation from engineering values to raw values and vice versa.

9.2 Starting the Calibration Editor

The Calibration Editor (CE) can be started by selecting the **Calibration Editor** button in the EuroSim start-up window (see Section 5.2.3).

9.3 Calibration types

There are three types of calibration:

- polynomial equation
- interpolation
- lookup table

The polynomial equation is a continuous function of the format

$$y = ax^4 + bx^3 + cx^2 + dx + e \tag{9.1}$$

The constants a,b,c,d,e are coefficients which, when correctly chosen, approximate any correlation function closely enough for the intended purpose.
The interpolation method uses point pairs to create a continuous function by performing a linear interpolation between these points.
The lookup table method creates a discrete correlation function using a lookup table to convert the input to the output value. If the input value is not present in the lookup table, an error condition is raised. (Thus similar to point pairs, but without linear interpolation).

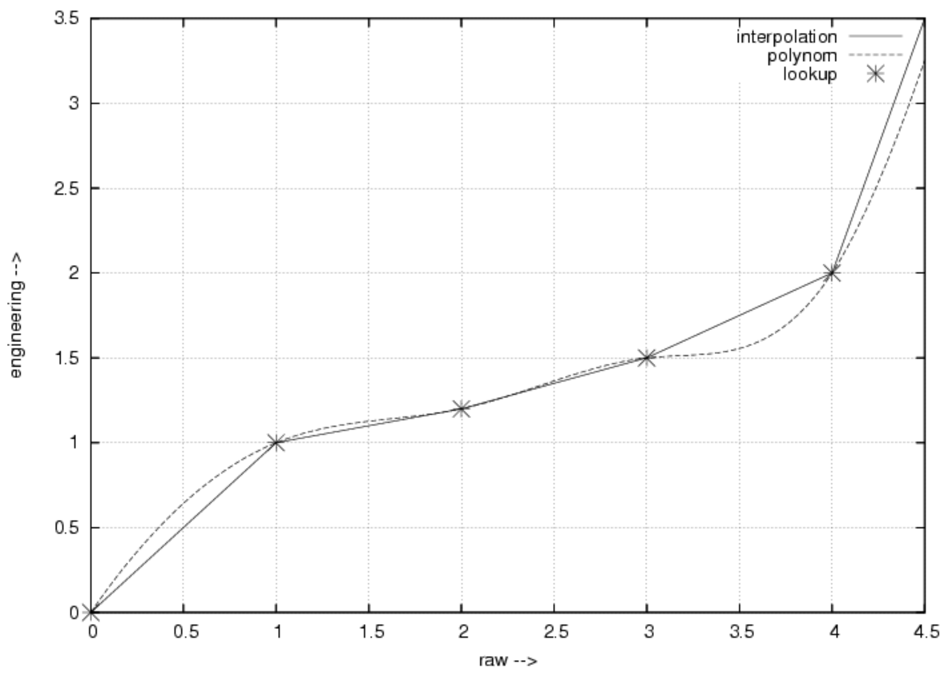
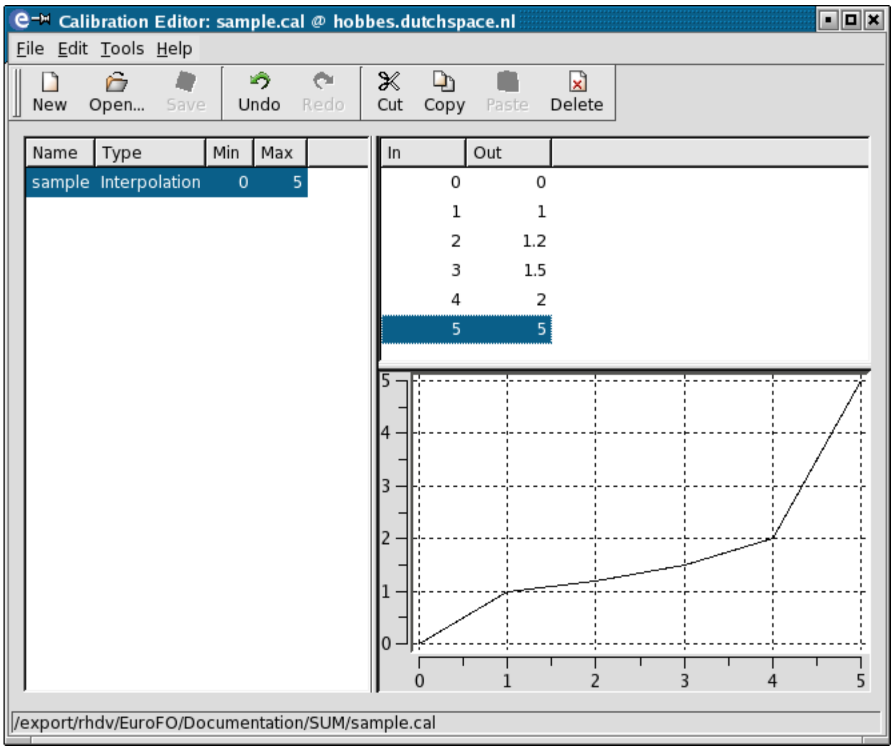


Figure 9.1: Calibration types


Calibration Editor: sample.cal @ hobbes.dutchspace.nl
File Edit Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete

Name	Type	Min	Max
sample	Interpolation	0	5

In	Out
0	0
1	1
2	1.2
3	1.5
4	2
5	5

/export/rhdv/EuroFO/Documentation/SUM/sample.cal

Figure 9.2: Calibration Editor

9.3.1 Calibration view

The calibration view provides an overview of the calibrations in the opened Calibration file.

9.3.2 Table view

The table view shows the data for a single calibration curve in tabular form.

9.3.3 Graph view

The graph view shows the data for a single calibration curve in a graphical form as a 2D curve.

9.4 Menu items

Note that most common commands are also available in context sensitive menus that pop-up when clicking the right mouse button. Some commands also have keyboard short-cuts.

9.4.1 Edit menu

New Calibration...
: Add a new calibration curve. This will show a dialog box to enter the name, type and min/max values of the new calibration curve.

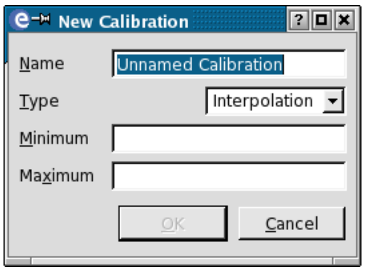


Figure 9.3: New Calibration dialog box

Add Data Row
: Add a new data row to the currently active calibration curve.

Rename
: Rename/start editing the first column of the row which has focus.

Delete Delete the selected rows in the currently active view.

Select All
: Select all rows of the currently active view.

9.5 Curve Restrictions

The following restrictions are applicable to data elements in each curve:

- No duplicate In/Power/Index values
- The lookup table must contain at least one entry
- The polynom must have at least one coefficient
- The interpolation must have at least two point pairs

9.6 Using Calibrations

9.6.1 Calibration API

The calibration API consists of only three functions. The first function is called `esimCalibrationLookup`. This function looks up a calibration curve given its name. The function returns a handle which can then be used to perform the actual calibration. To perform the actual calibration you call `esimCalibrate`. The function takes the calibration handle and a value to calibrate and produces the calibrated value and a status code.

Detailed information can be found in the on-line manual page esimCalibration and in the EuroSim Manual Pages document.

9.6.2 Selecting Calibration Files

When starting a simulator a user must specify which calibration files to use for a specific run. This is done in the Simulation Controller. On the Input Files tab there is a section called Calibrations. You can add or delete the calibration files you want to use.

Chapter 10

SMP2 Editor reference

This chapter provides details on the SMP2 Editor. The various objects which can be edited, the menu items of the editor, and their options are described. For menu items not described in this chapter, refer to Section 3.5.
Simulation Model Portability (SMP) is ESA's standard for simulation interfaces. The purpose of the standard is to promote portability of models among different simulation environments and operating systems, and to promote the re-use of simulation models. EuroSim has implemented an interface for this standard.
SMP2 is the successor of SMP. SMP2 is a totally new standard, adopting state-of-the-art techniques, and has a much wider scope than its predecessor. The way of working with this standard and its complexity demand tools for specification, development, integration, and storage of the SMP2 models. EuroSim has an SMP2 Editor to edit SMP2 catalogues, code generation tools to generate C++ model code from SMP2 types, and an interface that integrates an SMP2 model with other EuroSim compatible models, e.g. SMP models.
Profound knowledge of the SMP2 standard is a prerequisite for successfully using the SMP2 Editor to create SMP2 models. For an overview of the standard, refer to [SMP05c]. For a comprehensive, formal description of the standard, see [SMP05e] for the SMP2 Meta Model (or Simulation Model Definition Language, SMDL), [SMP05b] for the SMP2 Component Model, [SMP05a] for the SMP2 C++ Mapping and [SMP05d] for the SMP2 Model Development Kit (MDK).

10.1 Using SMP2 in the EuroSim Environment

With the current release of EuroSim, a subset of the SMP2 standard is supported. See Section 10.6 for details on compliance. This subset includes the key functionality for developing SMP2 models and integrating them in a EuroSim simulation. Specifically, EuroSim supports only the Catalogue document type. The entire work-flow from creating catalogues to compilation and integration into a EuroSim simulator has been fully automated[1] using the following:

- SMP2 Editor

  The graphical tool fully integrated in the EuroSim tool-set that allows to create and edit SMP2 catalogues. No separate catalogue validation tool is necessary: only valid catalogues can be created using this tool. Powerful undo/redo functionality allows correcting mistakes.

- SMP2 Validator **smp2val**

  This utility allows validation of SMP2 catalogues from the command line. It is not mandatory when using the SMP2 Editor.

- SMP2 Code Generator **smp2gen**

  In order to successfully use SMP2, the time-consuming and error-prone process of producing code according to the SMP2 C++ mapping needs to be automated. This utility allows generation of

[1] Of course, model logic has to be hand coded.

code from a catalogue. Generated header files are compliant with the standard's C++ mapping. Generated implementation files separate boilerplate code from the model logic. Boilerplate code is based on the SMP2 MDK.

The code generation utility is integrated in the SMP2 Editor and also available from the command line for tweaking of the code generation parameters.

The code generator is automatically invoked from the ModelEditor when building the simulator executable. It generates the boilerplate code and the header files into the directory where also the object files are stored. The user only has to add the implementation files to the model file. These implementation files must be generated by the code generator.

Example:

```
smp2gen -c sensor_model -i spacecraft.cat
```

- EuroSim SMP2 assembly file

  The EuroSim replacement of the SMP2 assembly file is a simple list of instances of models.

  The format is as follows. Each line contains a list of instances. It starts with the name of the catalogue file, followed by the fully qualified name of the model and finally followed by a comma separated list of instances. Comments are allowed anywhere and start with a # sign.

  The fully qualified name of the model is the name of the model in the namespace hierarchy. The hierarchical elements are separated by forward slashes.

  Example:

```
# Catalogue       Model              Instance(s)
spacecraft.cat    /SC/Sensors/GPS    GPS1, GPS2, GPS3
spacecraft.cat    /SC/AOCS           AOCS
spacecraft.cat    /SC/Sensors/STR    STR1, STR2, STR3
ground.cat        /ground/station    station1, station2
```

  This file is automatically processed by a code generator at build time. You must add at least one such file to your model file or else no instances of models will be created at all. More than one EuroSim assembly file may be specified.

- Integration in Model Editor including automatic building of simulators

  As this version of EuroSim does not support the SMP2 assembly files, a replacement facility has been implemented to allow users to instantiate a number of models from catalogue files in the model editor.

  The user must include the catalogue files used in the EuroSim assembly file in the model file. The EuroSim build system automatically generates all the source code files needed to create the simulator. The implementation code must be provided by the user.

- Integration of SMP2 models and non-SMP2 models

  Use the EuroSim Model Editor to add EuroSim compliant models to the generated SMP2 model file and the Model Description Editor to make the published fields available for non-SMP2 models. Refer to Chapter 7 for more information on the Model Description Editor. The Parameter Exchange Editor (see Chapter 8) can be used to add dataflows between models (a feature currently not available from within SMP2 models). SMP2 entry points can be scheduled by the Schedule Editor (see Chapter 11) just like ordinary EuroSim entry points. The SMP2 schedule specification is not suitable for real-time simulation.

Apart from the tool and utilities described above, the EuroSim distribution comes with:

- C++ and XML sources of the supported version (v1.2) of the standard.

- Compiled versions of the MDK (debug and non-debug)
- A compiled version of the Component Model library that allows running of SMP2 models in the EuroSim run-time environment.

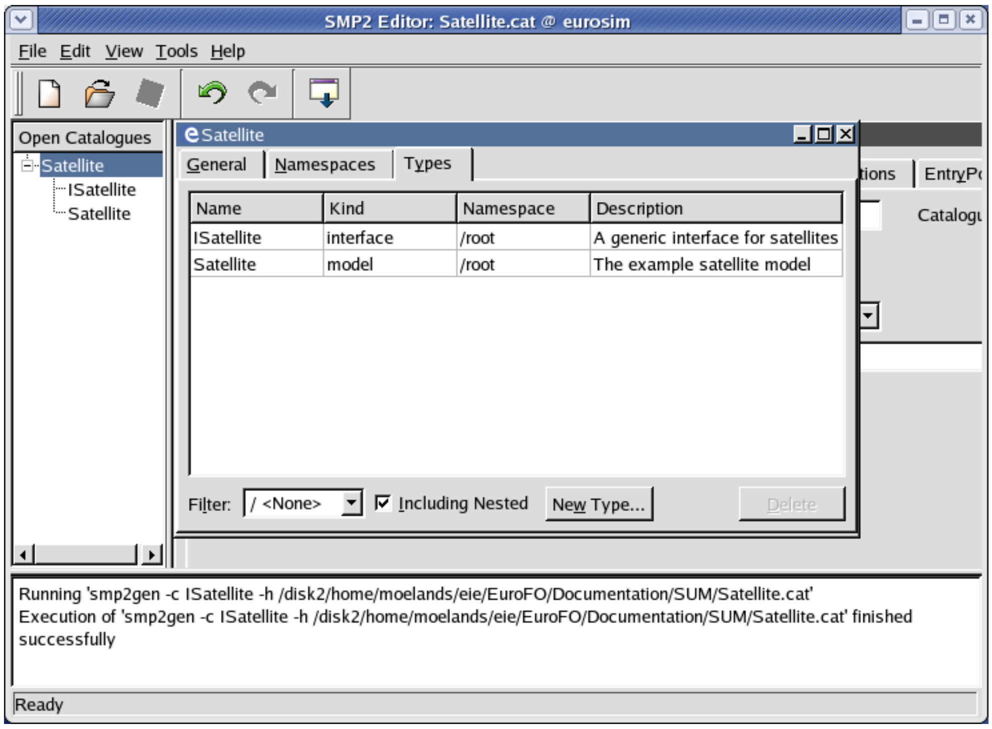
For the utilities described, on-line manual pages [MAN05] are available.
Summarizing, while being compatible with the SMP2 standard, EuroSim's SMP2 support is aimed at full integration of developed models in the EuroSim environment, aiming at making available EuroSim's real-time simulation capabilities towards the SMP2 community, while allowing the re-use of native EuroSim and SMP real-time models. The EuroSim user will have little trouble learning how to use EuroSim's SMP2 capabilities as they are fully integrated in the EuroSim toolset and way of working.

10.2 Starting the SMP2 Editor

The SMP2 Editor can be started from the Model Editor. When the model tree contains a file with the appropriate extension (see Appendix F), then the SMP2 Editor is automatically started when the Edit command is selected on the file node in the Model Editor.

10.3 SMP2 Editor Overview

The SMP2 Editor is centered around a so-called multiple document interface (MDI) that allows displaying multiple windows containing information on SMP2 objects.
The main SMP2 meta model elements have their own window in the SMP2 Editor. These elements are the catalogue itself, and the types contained in it. The various pieces of information that make up the catalogue or type are grouped in tabs.
On the left side of the SMP2 Editor window, the currently open catalogues and the types they contain are displayed in a hierarchical view. This allows easy selection. On the MDI that covers the main part of the screen, the windows of the selected SMP2 element is shown on top.
The bottom part of the screen is dedicated to the output of the code generation process.



Name	Kind	Namespace	Description
ISatellite	interface	/root	A generic interface for satellites
Satellite	model	/root	The example satellite model

```
Running 'smp2gen -c ISatellite -h /disk2/home/moelands/eie/EuroFO/Documentation/SUM/Satellite.cat'
Execution of 'smp2gen -c ISatellite -h /disk2/home/moelands/eie/EuroFO/Documentation/SUM/Satellite.cat' finished
successfully
```

Figure 10.1: Overview of the SMP2 Editor screen

10.3.1 General remarks on editing a catalogue

The SMP2 Editor ensures that at all times, a catalogue is valid. This means that no invalid catalogue files can be produced, and validating a created catalogue file is not necessary. The editor enforces validity by doing all necessary semantic checks on every edit action of the user, and taking measures if necessary. This way of working provides powerful editing possibilities.
As an example, suppose the user deletes a namespace which contains types. Apart from the namespace itself, all types in this namespace and all nested namespaces will be deleted. Moreover, if any types located in other namespaces make use of a deleted type, it will be either modified or deleted, depending on the nature of the using type. E.g. a field may be removed from a structure if the type of the field is deleted. An array will be deleted if its item type is deleted.
Another example is the modification of an array's item type. It will not be allowed by the editor to create a recursive type system. Only those item types that will not create a recursive type system can be chosen from. So, the array type itself may not be chosen as item type, nor may a structure that contains a field that has the array as its type.
As another example, names must be unique in their context. E.g. the names of all nested namespaces and types which are defined in a certain namespace, must be unique. On creating a new object or renaming an existing object, the SMP2 Editor automatically checks if the new name is unique for its context. If not, it is made unique by automatically appending an index number to the name.
If the user is not satisfied with the result of an edit action, the SMP2 Editor's undo functionality can restore up to 10 edit actions.

10.4 SMP2 Editor objects

10.4.1 Catalogue

The Catalogue window consists of three tabs.

General
General, textual information.

Namespaces
The tree of namespaces. In new catalogues, only a root namespace (named '/') is shown. Although a root namespace is not part of the SMP2 standard, here serves as a bootstrapping mechanism for adding new namespaces. The root namespace is special: it cannot be removed or renamed, and no types may be placed in it.

Add a namespace by right-clicking on the parent and selecting **Add Nested** from the context menu.

Note that no types are shown in the namespaces. They are listed under the *Types* tab.

Types Key information on all types defined in the catalogue. Note that types are not shown per namespace, but in a flat list instead. This provides a better overview.

A filter may be used to select the range of types to be shown. Selection may be done per single namespace or per namespace tree. **Filter** allows selection of a namespace. Check **Including Nested** to shown types from entire tree rooted at **Filter**. By default, all types are shown (in that case, **Filter** is '/', the root namespace).

The built-in SMP2 types are never shown in this list.

This list is read only. Double clicking an entry in the list opens the type's window and allows viewing and editing of all information. Each kind of type has its own window type.

To add a new type, click **New Type...**. This opens the New Type dialog.

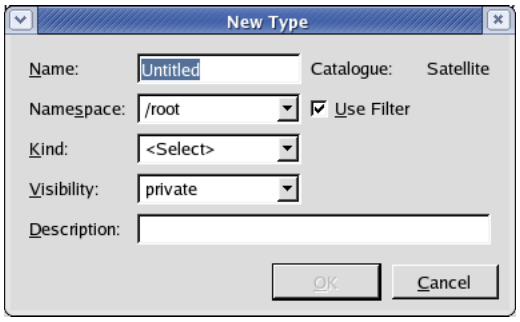


Figure 10.2: New Type dialog

This dialog allows setting key information on the new type. This information is the same for all kinds of types. After clicking **OK**, the new type's window will be opened for further editing of the newly created type.

No types can be created if no namespace is added to the root namespace.

10.4.2 Types

Each type kind has its own window layout. However, some common information is always shown in a *General* tab. This is the key information that the user entered when creating a new type. Most of the fields can still be edited after creation of the type, with the exception of **Namespace** and **Kind**.

10.4.3 Integer

Apart from the *General* tab, the window for the **Integer** type has a single tab *Details* which allows setting the lower and upper bounds.

10.4.4 Float

Apart from the *General* tab, the window for the **Float** type has a single tab *Details* which allows setting the lower and upper bounds, including the **Inclusive** flags, and the **Unit**.

10.4.5 Enumeration

Apart from the *General* tab, the window for the **Enumeration** type has a single tab *Literals* which allows creating a list of literals. Enter the name of the new literal and press **Add** to add a new literal. Select a literal from the list and press **Delete** to remove it again.
To be a valid enumeration, at least one literal must be present. The editor however allows creation of an 'empty' enumeration type, assuming the user will add literals once it has been created. On saving a catalogue containing an empty enumeration, a dummy literal will be inserted in the file to make it valid. This dummy will not be inserted in the opened enumeration.
By double clicking one of the fields in the table of literals, the field can be edited.

10.4.6 String

Apart from the *General* tab, the window for the **String** type has a single tab *Details* which allows setting the string size. On creation, a default size is provided.

10.4.7 Array

Apart from the *General* tab, the window for the **Enumeration** type has a single tab *Details* which allows setting the array size and item type. On creation, a default size and item type are provided.

10.4.8 Event

Apart from the *General* tab, the window for the **Event** type has a single tab *Details* which allows editing the optional argument type.

10.4.9 Structure

Apart from the *General* tab, the window for the **Structure** type has a single tab *Fields* which allows editing the list of fields and their attributes. New fields can be added and existing fields can be modified and deleted.

10.4.10 Class

Apart from the *General* tab, the window for the **Class** type has the following tabs.

Inheritance
Allows setting the single base class and to define the class as abstract.

Fields Identical to the *Fields* tab of the **Structure** window.

Properties
Allows editing the properties list of the class.

Operations
Allows editing the list of operations of the class, and their parameters. Operations may be viewed and defined in the upper half of the window. By selecting one of the operations from the list, the parameters of the operation may be viewed and edited in the bottom half of the window.

Associations
Allows editing the associations of the class.

10.4.11 Interface

Apart from the *General* tab, the window for the **Interface** type has the following tabs.

Inheritance
Allows setting the list of base interfaces.

Properties
Identical to the *Properties* tab of the **Structure** window.

Operations
Identical to the *Operations* tab of the **Structure** window.

10.4.12 Model

Apart from the *General* tab, the window for the **Model** type has the following tabs.

Inheritance
Allows setting the single base model.

Properties
Identical to the *Properties* tab of the **Structure** window.

Fields Identical to the *Fields* tab of the **Structure** window.

Interfaces
Allows setting the list of implemented interfaces.

Operations
Identical to the *Operations* tab of the **Structure** window.

Associations
Allows setting the list of associations.

Entry Points
Allows setting the list of entry points. On the left side of the window, the entry points can be viewed and edited. When selecting an entry points from the list, on the right side of the windows the input and output fields can be viewed and edited.

Event Sources
Allows setting the list of event sources.

Event Sinks
Allows setting the list of event sinks.

References
Allows setting the list of event references.

Containers
Allows setting the list of containers.

10.5 Menu items

10.5.1 File menu

New Creates a new empty catalogue

Open Opens a catalogue. It is possible to have multiple catalogues open. If the catalogue being opened has *xlinks* to other catalogues, they will be opened as well.

Close Closes a catalogue

Save Save the current catalogue. If other catalogues are *xlinked*, it is advisable to use the *Save All* command instead.

Save As Rename and save the current catalogue. This function must also be used when copying a catalogue. Moving or copying a catalogue from the UNIX prompt will not work. Note, that any *xlinks* to the renamed catalogue from other catalogues that are currently open will be changed to the new catalogue name.

Save All
Save all open catalogues that need saving.

10.5.2 Edit menu

Undo/Redo
Undo/Redo edit actions. Up to 10 levels of undo/redo are available.

Cut/Copy/Paste
Currently not available.

Delete Delete the current selection.

Rename
Rename the current selection. Any *xlinks* to the renamed catalogue or type from catalogues that are currently open will be changed to the new name. Note that renaming can also be done by editing the **Name** attribute in the *General* tab of a catalogue or type window.

10.5.3 View menu

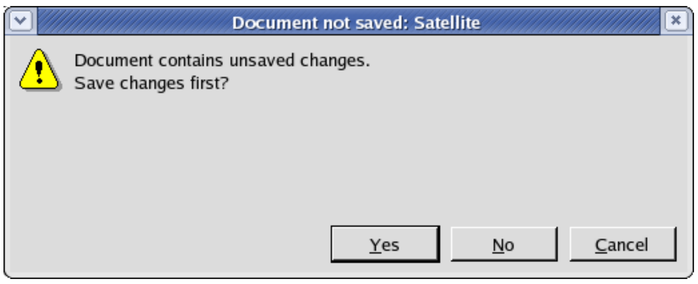
Large toolbar buttons
Selecting this option shows large buttons in the toolbar.

Toolbar button labels
Selecting this option shows labels or toolbar buttons.

10.5.4 Tools menu

Generate
Generate code for the current selection. This can be for a catalogue or for an SMP2 type that results in a C++ class (Class, Interface, and Model).

Code can only be generated for catalogues that are saved. Therefore, if the current selection is (part of) a catalogue that is not saved, you will first be prompted to confirm saving the catalogue.


Document not saved: Satellite

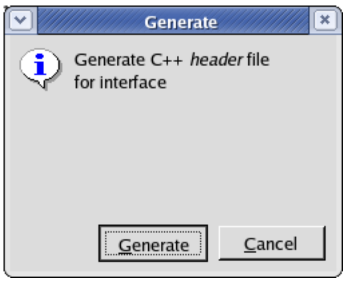
Document contains unsaved changes.
Save changes first?

Figure 10.3: Your are prompted to confirm saving the catalogue before generating code

The following files can be generated:

- a C++ mapping compliant header file for the catalogue containing all type declarations that are not C++ classes
- a C++ mapping compliant header file for each type that results in a C++ class (Class, Interface, and Model)
- a boilerplate file for each Class and Model, implementing the class's interface
- an implementation file for each Class and Model, allowing the model developer to add model logic.

If the current selection is a catalogue, a type that will not result in a generated C++ class, or an Interface, only a header file will be generated. You will be prompted for to confirm this.

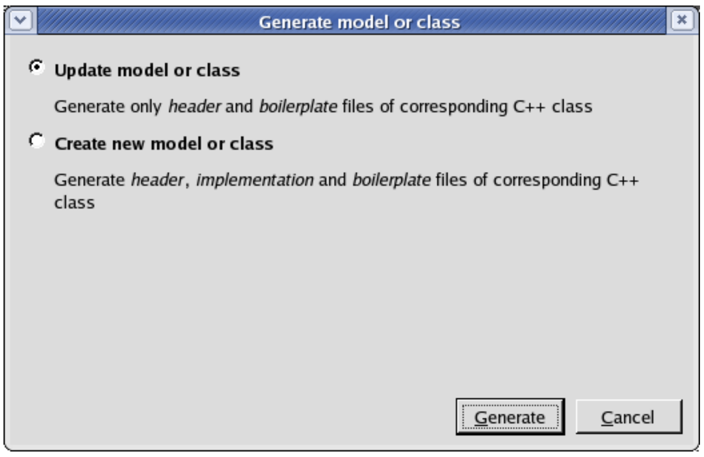

Generate

Generate C++ *header* file
for interface

Figure 10.4: Your are prompted to confirm the generation of a header file

Note that no files are overwritten. Instead, if earlier generated files exist, they are saved in ***filename*.old**.

If, on the other hand, a Class or Model is currently selected, you are asked to make a choice between generating all files (header, boilerplate, implementation) or to update only header and boilerplate (keeping an existing implementation file, presumably with model logic code added by the user).



Update model or class [✓]

Generate only *header* and *boilerplate* files of corresponding C++ class

Create new model or class

Generate *header*, *implementation* and *boilerplate* files of corresponding C++ class

Figure 10.5: Your are prompted to select the files to be generated

Note that an updated Model may need regeneration of its implementation file (e.g. if entry points are added).

As an alternative for these code generation options, the **smp2gen** tool may be called directly from the command line.

10.6 SMP2 Compliance

This section describes EuroSim's compliance to the standard, grouped per SMP2 document. Note that the Handbook [SMP05c] is non-normative.

10.6.1 Applicable SMP2 version

The implementation is based on SMP2.0 version 1.2 (release date 28 October 2005).

10.6.2 Metamodel

- Only catalogues are currently supported. Assemblies and schedules are not supported. For creating configurations of model instances, use the **smp2glue** utility. For creating real-time schedules, use the EuroSim Schedule Editor. SMP2 schedules are not suitable for creating the hard real time simulators that form EuroSim's core functionality.
- Metadata and its subclasses Comment, Documentation, and Attribute, used to annotate SMP2 elements, are not supported. Use e.g. the **description** field to annotate catalogues and types.
- Visibility (part of Visibility Element) is implemented, but its semantic constraints are ignored for dependencies between Visibility Elements. This means e.g. that types can use other types independent of their visibility. Public visibility is assumed here. The SMP2 Editor allows setting the visibility however, and it is taken into account during code generation.
- Native Type and Platform mapping, used by Native Type, are not supported. Primitive Type is supported, although not based on Native Type.
- Unique values of Enumeration Literal are not enforced in the editor. It is checked on reading catalogues, however.
- The Primitive Type of Integer is assumed to be Int32.

  The Primitive Type of Float is assumed to be Float64.

- Value and its subclasses are not supported.
- Attribute Type and Attribute are not supported.
- Property's Category is not supported.
- Types nested in a Model are not supported.
- DefaultModel of Container is not supported in the editor, and otherwise in a limited way.
- AttachedField of Property is not supported in the editor.
- The Minimum, Maximum, and Operator default attributes are not supported. The Smp.Attributes namespace is not supported. The OperatorKind enumeration type is not supported.
- The following features are part of the *IManagedModel* interface and are not supported by the EuroSim Component Model: Operations, Associations, Event Sources, Event Sinks, and References.

10.6.3 Component model

- EuroSim uses a static mechanism for publication of variables and entry points, i.e. at simulator build time all publication code is generated and a data dictionary containing variables and entry points is created. SMP2 on the other hand creates a type system, model fields, and entry points at run-time. This dynamic mechanism is mapped to EuroSim's static one by running, during the building of the simulator, the SMP2 compiled model code linked with a special version of the Component Model that gathers information and converts it to the EuroSim format. After linking the model code with the run-time Component Model, the simulator is ready for running in the EuroSim environment. SMP2 model fields are mapped to EuroSim variables. SMP2 entry points are mapped to EuroSim entry points. SMP2 operations are not supported in the EuroSim run-time interface.
- The optional self persistence interface (*IPersist* and friends) is not supported. External persistence (by EuroSim) is used.
- The optional Dynamic Invokation interface (*IDynamicInvokation* and friends) is not supported.
- The optional *IDynamicSimulator* and *IFactory* interfaces are not supported.
- The SMP2 *IScheduler* interface is quite limited in its possibilities to define a schedule for real-time applications. Use the EuroSim Schedule Editor instead to create schedules for real-time purposes. SMP2 entrypoints can be scheduled just like normal EuroSim entrypoints.
- Some convenience methods in the *IPublication* interface are not implemented.

  PublishArray() and *PublishStructure()* are not supported. Publish the type first and use *PublishField()* instead. *PublishOperation()*, *GetFieldValue()*, *SetFieldValue()*, *CreateRequest()*, *DeleteRequest()*, *SetArrayValue()*, *GetArrayValue()* are not implemented.
- *IPublishOperation* is not implemented.
- *IPublication::PublishProperty* is not implemented.
- The *view* parameter of the *IPublication::PublishField* methods is ignored by EuroSim.
- The *minIncluse* and *maxInclusive* parameters of *ITypeRegistry::AddFloatType* are ignored by EuroSim.
- *IType::Publish()* is not implemented. Use the *IPublication* interface instead.
- The optional *IResolver* service is not supported.
- The EuroSim scheduler states are mapped to SMP2 states as follows:

EuroSim	**SMP2**
Unconfigured	Building, Connecting, Initialising
Initializing	n/a
Standby	Standby
Executing	Executing
Exiting	Exiting
Aborting	Abort

Table 10.1: EuroSim scheduler state to SMP2 state mapping

The SMP2 ISimulator state change commands are mapped to EuroSim state changes as follows:

SMP2 state change commands	**EuroSim state**
Run	Executing
Hold	Standby
Store	Not Implemented
Restore	Not Implemented
Exit	Exit
Abort	Abort

Table 10.2: SMP2 ISimulator state change mapping to EuroSim states

10.6.4 C++ mapping

- The generated code is human readable.
- It is possible to generate all header code in a single file. Alternatively, the code for each C++ class can be generated to a separate file.
- It is possible (and recommended) to generate each type surrounded by its own namespace declaration, or alternatively group all types in a single namespace declaration.
- On top of the C++ mapping which concerns only header files, separate implementation and boilerplate files can be generated that automate writing of model code as much as possible.
- Parameterless operations cannot be supported due to a known bug in SMP2's MDK.
- In-line property getters and setters are supported as suggested in the standard.
- Support for the Static flag of Feature in boilerplate file is limited. Hand coding may be necessary in boilerplate files.
- A public *static void Publish(IPublication*)* method is generated for SMP2 Classes that may be used to publish the class's fields.
- An implementation of the *Publish()* method for models is provided in boilerplate code. This method publishes all types that may be used in the model and all the models' fields and operations. Although an attempt is made to publish types in the correct order, in case of complicated types it may be necessary to rearrange this generated code for correct order of publication of types.

- Four methods have been added to a model's interface: *UserCreate()*, *UserDelete()*, *UserConnect()*, and *UserInitialize()*. Their empty bodies are generated as part of the C++ class's implementation. Logic may be provided by the model developer. Implementations for a model's constructors and destructor are generated that call methods *UserCreate()* and *UserDelete()*, respectively. An implementation of the *Initialize()* method for models is generated that calls method *UserInitialize()*. An implementation of the *Connect()* method for models is generated that calls method *UserInitialize()*.

- Generated model code is fully SMP2 compliant. *IManagedModel*, *IAggregate*, *IEventProvider*, *IEventConsumer*, and *IEntryPointPublisher* interfaces are always inherited, even when not strictly necessary. See section Section 10.6.3 for limitations to the Component Model that may imply limitations to functionality of the generated model code, if use in combination with the EuroSim Component Model implementation (e.g. model functionality that uses the *ISimulator* interface internally).

10.6.5 Model development kit

Generated code as well as the Component Model library are based on the MDK ([SMP05d] where applicable.

Chapter 11

Schedule Editor reference

This chapter provides details on the Schedule Editor. The various items which can be placed on the schedule tab pages, all menu items of the editor and their options are described. For menu items not described in this chapter, refer to Section 3.5.

11.1 Starting the Schedule Editor

The Schedule Editor can be started by selecting the 'Schedule Editor' button in the Project Manager window or by choosing the *Tools:Schedule Editor* menu item. If no schedule file is selected in the Project Manager tree view, the Schedule Editor starts with a new schedule. It is recommended to use a filename of the form *modelname*`.sched`. The Schedule Editor can also be started by double clicking a schedule file in the 'Files' list of the Project Manager. When creating a new schedule, the Schedule Editor automatically uses the name of the model file that is currently selected in the Project Manager.

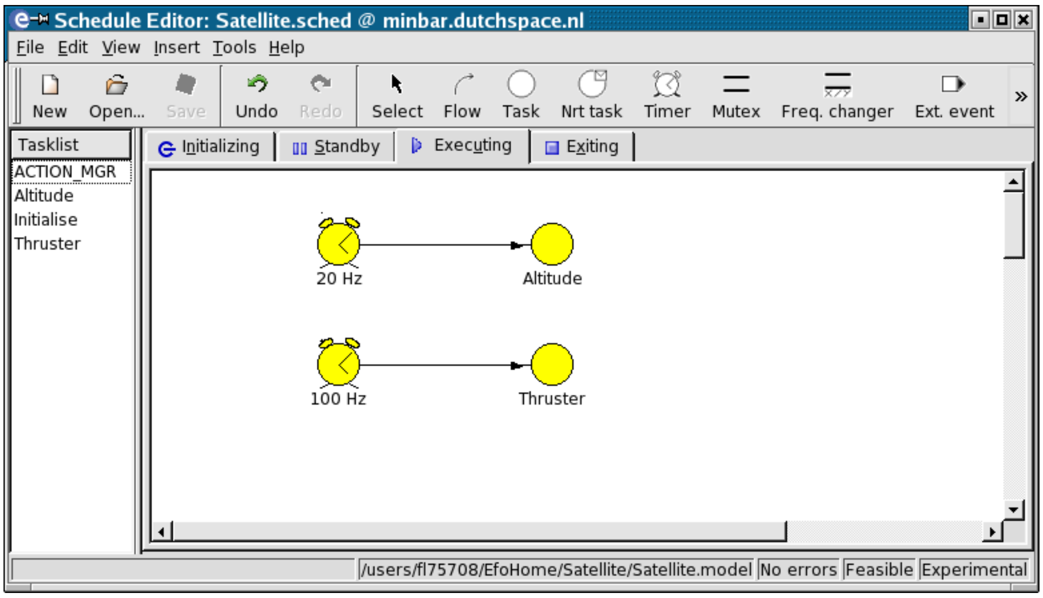

Schedule Editor: Satellite.sched @ minbar.dutchspace.nl
File Edit View Insert Tools Help
New Open... Save Undo Redo Select Flow Task Nrt task Timer Mutex Freq. changer Ext. event
Tasklist
ACTION_MGR
Altitude
Initialise
Thruster
Initializing Standby Executing Exiting
20 Hz
Altitude
100 Hz
Thruster
/users/fl75708/EfoHome/Satellite/Satellite.model
No errors
Feasible
Experimental

Figure 11.1: Schedule Editor window

11.2 Schedule Editor items

In the Schedule Editor tab pages, a schedule can be created by positioning schedule items (tasks, mutual exclusions, frequency changers, internal and external events, output events, timers) and connecting them with flows. A schedule is a set of attributed tasks, timers, scheduling events and their respective dependencies. The overall behavior of a schedule is deterministic, whereas that of a single task need not be.

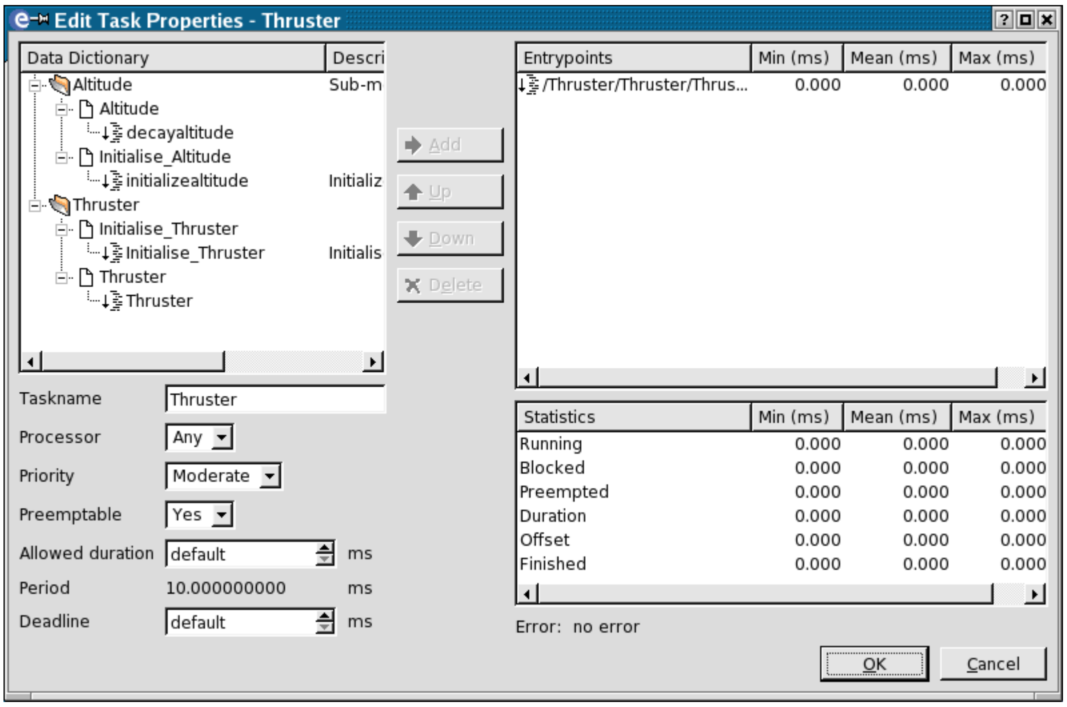
When an item is placed in the tab page, it is given some default values for the properties of the item. These can be changed by double-clicking the item, or by selecting the item and activating the menu item *Edit:Properties* (or pressing Alt-Enter on the keyboard). When the item is shown in a color other than yellow, there is an error for the item. The error message can be viewed alongside the properties of the item. For a list of possible error messages, refer to Appendix C.
Items in the tab page can be repositioned by selecting the item with the left mouse button and, whilst holding the button pressed down, moving the item to another location on the tab page. All flows to and from the item will remain connected.
Labels can also be repositioned in the same way. This allows you to move the label out of the way if a flow passes through the label. The position of the label remains relative to the item it belongs to.
In the next sections, each of the items is described, together with the properties which can be modified. The graphic representation of the item in the tab page of the Schedule Editor is shown on the left.

11.2.1 Tasks

A task item represents a list of one or more entrypoints. Each task represents a single execution unit during the simulation. Grouping entrypoints within a task will ensure that the operations (represented by the entrypoints) are executed sequentially. In a schedule, tasks can be activated by:

- a simulator execution state transition (STATE_ENTRY connector on entering and STATE_EXIT connector on leaving a state)
- completion of another task
- periodically, using a timer which triggers the task at a given frequency
- through an input connector that is triggered from an operation that has ended execution
- a frequency changer

A task can be used in more than one state.


Edit Task Properties - Thruster

Data Dictionary	Descri
Altitude	Sub-m
Altitude	
decayaltitude	
Initialise_Altitude	
initializealtitude	Initializ
Thruster	
Initialise_Thruster	
Initialise_Thruster	Initialis
Thruster	
Thruster	

Entrypoints	Min (ms)	Mean (ms)	Max (ms)
/Thruster/Thruster/Thrus...	0.000	0.000	0.000

Taskname	Thruster
Processor	Any
Priority	Moderate
Preemptable	Yes
Allowed duration	default ms
Period	10.000000000 ms
Deadline	default ms

Statistics	Min (ms)	Mean (ms)	Max (ms)
Running	0.000	0.000	0.000
Blocked	0.000	0.000	0.000
Preempted	0.000	0.000	0.000
Duration	0.000	0.000	0.000
Offset	0.000	0.000	0.000
Finished	0.000	0.000	0.000

Error: no error

Figure 11.2: Task dialog

The following properties can be modified in the Edit Task Properties window (see Figure 11.2):

Entrypoints
: This list shows all entrypoints that are associated with the task. The 'Data Dictionary' list contains all known entrypoints, the 'Entrypoints' list shows the entrypoints selected for the current task. The list can be modified by pressing the buttons in-between the two listboxes. An entrypoint can be copied from the 'Data Dictionary' list to the 'Entrypoints' list (right arrow), or removed from the task list (the 'Delete' button). The up and down arrow buttons can be used to re-order the entrypoints. For editing the entrypoint list a model file should be selected, so a data dictionary will be loaded into memory (see also Section 11.3.1): the data dictionary file of the model must have been build, otherwise the list will be empty and no entrypoints can be selected.

  Timing information for the selected entrypoint is shown next to the 'Entrypoints' list. Timing information can be modified by clicking on the entrypoint timing values. Timing information can also be imported into the scheduler using the *File:Import timings...* menu item. The latter is only possible if you have already performed a simulation run with this schedule, which produces the timings file.

  Beneath the entrypoint values the total timings for the current task are displayed. Entrypoints in a task are executed sequentially, so the timing information is calculated by adding the values for the individual entrypoints in the task.

Taskname
: The name of the task.

Processor
: The processor on which the task should be executed. The default is 'Any'.

Priority The priority with which the task should run. Default is 'Moderate'.

Preemptable
: Set this to 'No' if the task may not be interrupted by another task.

Allowed Duration
: The maximum time a task may take during simulation. This sets the deadline on which the task should have completed. The deadline is calculated by adding the Allowed Duration to the simulation time at the moment a task becomes ready for execution. This attribute is only available for periodic tasks. The allowed execution time has to be equal to or less than the period.

Deadline
: The time in which this task should be finished, measured from the start of the current cycle. The default deadline is the end of the cycle.

Times (for *Allowed Duration* and *Deadline*) are always in multiples of the basic clock cycle (see Figure 11.8).
Task statistics are shown in the window below the entrypoints:

Running
: The time that the code in the entrypoints was actually executing.

Blocked
: The time between task activation and start of execution.

Preempted
: The time the task was preempted by a higher priority task.

Duration
: The total time to execute the task entrypoints.

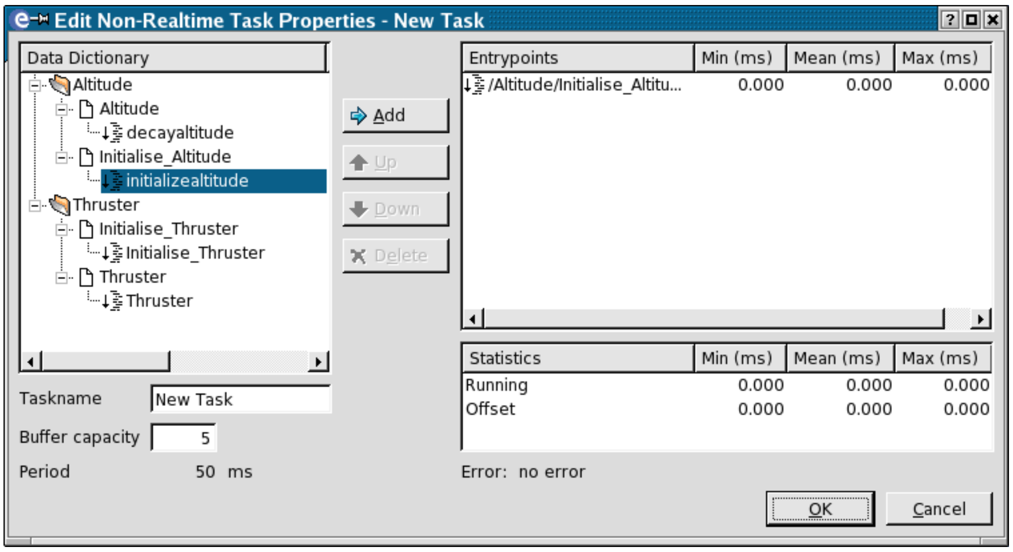
Offset The start of execution measured from the start of the current cycle.

Finished
: The end of execution measured from the start of the current cycle (Offset + Duration).

The last item, *Error*, shows the status of the item.

11.2.2 Non real-time tasks

Non real-time tasks are the links between the real-time domain and the non real-time domain. A non-real-time task can be raised by a completed task, by an internal event or by an external event.
When the schedule is executed by the scheduler, all tasks (seen as a set of entrypoints) connected to a non real-time task will be executed in the non-real time domain. For each activation of the non real-time task this will be done once, unless the buffer overflows because tasks in the non-real time domain can not be executed fast enough.



Entrypoints	Min (ms)	Mean (ms)	Max (ms)
/Altitude/Initialise_Altitu...	0.000	0.000	0.000

Statistics	Min (ms)	Mean (ms)	Max (ms)
Running	0.000	0.000	0.000
Offset	0.000	0.000	0.000

Taskname	New Task
Buffer capacity	5
Period	50 ms

Error: no error

Figure 11.3: Non Real-time Task Dialog

The following properties can be modified in the properties dialog (see Figure 11.3)

Entrypoints
This field indicates the entrypoints that will be triggered by this non real-time task. This list can be modified just like real-time tasks (see Section 11.2.1).

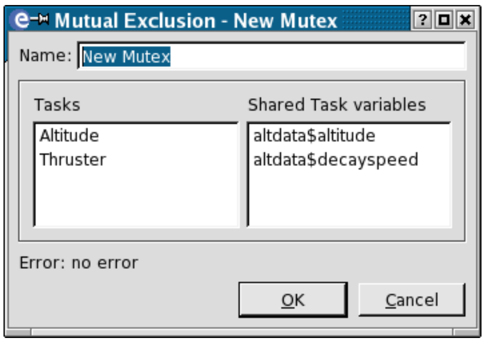
Taskname
The name of the non real-time task.

Buffer Capacity
This indicates the buffering capacity of the non real-time task.

The *Period* field is inherited from the schedule. *Timingsfile* shows the selected timingsfile. *Error* shows the status of the non real-time task.

11.2.3 Mutual exclusions

Mutual exclusions are used for asynchronous stores. Independently of the direction of a connected flow, only one task (of those connected to the store) will be executed at a time. The sequence of execution is done on a first-come first-serve basis.



Name: New Mutex

Tasks	Shared Task variables
Altitude	altdata$altitude
Thruster	altdata$decayspeed

Error: no error

Figure 11.4: Mutual Exclusion Dialog

The following properties are shown in the properties window (see Figure 11.4):

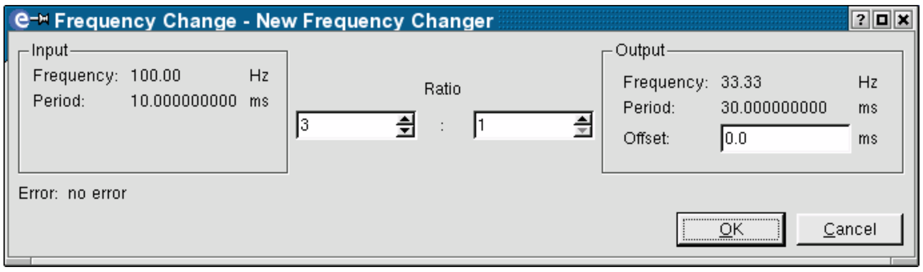
Tasks This list shows all tasks currently connected to the mutual exclusion.

Shared Task Variables
The Shared Task variables box shows a list of the variables that are shared by the listed task(s).

The last item, *Error*, shows the status of the item.

11.2.4 Frequency changers

Frequency changers, or synchronous stores, are used for multiple frequency dependencies, meaning that they transform the frequency of the incoming triggers into the store to another frequency going out of the store. Only one input connector is allowed for a frequency changer.



Input
Frequency: 100.00 Hz
Period: 10.000000000 ms

Ratio: 3 : 1

Output
Frequency: 33.33 Hz
Period: 30.000000000 ms
Offset: 0.0 ms

Error: no error

Figure 11.5: Frequency Change Dialog

The following properties can be modified in the properties window (see Figure 11.5):

Input Ratio and Output Ratio
show the ratio between the input and output frequencies. Only M:1 or 1:N ratios are allowed. An 1:N store (e.g. 10Hz/50Hz) means that upon activation of the frequency changer the output flows of the store are activated N times (5 in the example) directly one after another. To achieve a more regular task activation (50 Hz in the example), the task after the output flow should also be connected to a 50Hz timer. An M:1 store will activate the output flow only once every M input activations.

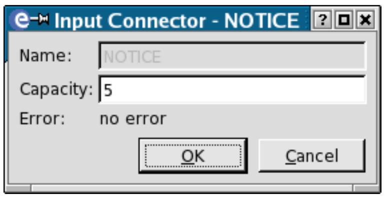
Offset The delay of the output activation in milliseconds. Only valid for M:1 ratios. It must be a multiple of the basic clock cycle (see Section 11.4.7). A value of zero (0) means that the output will be activated on the first input activation. The default activates the output after M input activations.

Note that the output side of the synchronous store runs mutually exclusive with the input side. See also Section 11.4.3 and Section 11.4.4.

The *Output Frequency* and *Output Period* are updated when the ratio changes.
The last item, *Error*, shows the status of the item.

11.2.5 Internal and External events

Internal and external events, both input connectors, represent events in the non-real time domain. An input connector activates its output flow when the event occurs. This may in turn execute a task or activate an output event. An internal event represents a predefined event related to simulator state changes and real-time errors. An external event is an event explicitly raised by the user from an MDL script or by an external event handler.


Input Connector - NOTICE

Name: NOTICE
Capacity: 5
Error: no error

Figure 11.6: Input Connector Dialog

The following properties can be modified in the properties window (see Figure 11.6):

Name The name of the input connector. Predefined events cannot be renamed, only user defined input events can be renamed. The name must be unique.

Capacity
This indicates the buffering capacity of the connector.

Raised by
This indicates the sources of the event. An event can be raised internally by model code, a script or the event connection. An event can also be raised by an External Event Handler, e.g. a handler connected to a HW device or a signal handler (see section Section 11.3.5: external event handler).

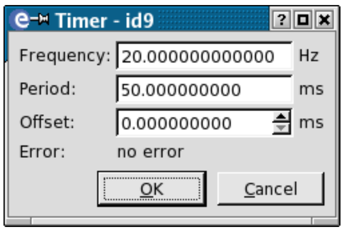
Error shows the status of the item.

11.2.6 Output events

An output connector can be raised by a completed task or by an input connector. It represents an event related to simulator state changes and scheduler mode switches.
A user defined output event activates the user defined input event that matches its name.
No properties can be modified. Only user defined output events can be renamed.

11.2.7 Timers

Timers activate their output at the specified frequency and can be used to activate f.i. tasks. The maximum allowed frequency can be defined in the Schedule Configuration tool (see Section 11.3.5). The system uses 200 Hz (IRIX) or 100 Hz (Linux, Windows NT) as default value.


Timer - id9

Frequency: 20.000000000000 Hz
Period: 50.000000000 ms
Offset: 0.000000000 ms
Error: no error

Figure 11.7: Timer Dialog

The following properties can be modified in the properties dialog (see Figure 11.7):

Frequency and Period
Use either of these to set the frequency of the timer. If one is modified, the other is updated automatically. The maximum and default frequency is 200 Hz (IRIX) or 100 Hz (Linux, Windows NT). The frequency range allowed is 0.001 Hz up to and including the maximum frequency, with a step 0.001 Hz.

Offset The delay of the output activation in milliseconds. This must be a multiple of the basic period (see Section 11.4.7).

Error shows the status of the timer.

11.2.8 Flows

Flows are used to connect items in the schedule. They represent triggers going from one item to another.

11.3 Menu options

11.3.1 File menu

Select Model
With this option, a different model file can be selected from a file selection window. If the model does not have a data dictionary built, then it is not possible to specify entrypoints for tasks and non real-time tasks.

Parameter Exchange files
Opens a dialog to view, add or remove Parameter Exchange files for the current schedule, see Chapter 8 on how to create parameter exchange files.

Import timings
With this option, a timings file can be imported into the schedule. A file selection window will be shown in which a file can be selected. Timings files are generated automatically by the simulator and importing one will overwrite any manually entered timing settings.

11.3.2 Edit menu

Rename
Opens an in-place line edit to rename the currently selected item.

Properties
Pop up a dialog in which the properties of the currently selected node can be edited. The same effect can be reached by double clicking on an item in the schedule tab page.

11.3.3 View menu

In this menu, the state whose schedule tab page should be raised to the top can be chosen. There are four possible states: *Initializing*, *Standby*, *Executing* and *Exit*.

Enlarge drawing area
Enlarges the drawing area so that more items can be placed. Note that printing the drawing area will resize it to fit all items on one page.

Shrink drawing area
Shrinks the drawing area.

Refresh Reads in the new data dictionary that is associated with the currently selected model. This option is useful if you have an instance of the Model Editor open and update the model - and data dictionary by building it - while you are also editing the schedule.

11.3.4 Insert menu

In this menu, an item can be found for each of the items described in Section 11.2. For the internal events and output events, a cascading sub menu is available, from which various predefined internal and output events can be selected. For an explanation of the predefined events, see Section 11.3.4.2 and Section 11.3.4.3.
When an item has been selected from this menu, the cursor will change to the selected item, after which the item can be positioned on the tab page. If a flow is chosen, click on the item from which the flow should go, keep the left mouse button pressed, move to the target item and release the mouse button.

11.3.4.1 External events

External event handlers that are of type 'automatic' automatically add their input connector to this menu. See Figure 11.3.5 on how to create an external event handler.

11.3.4.2 Predefined internal events

The following internal events are predefined:

NOTICE
: This event is raised when the `esimMessage()` or `esimReport()` with the `esimSeverity` parameter set to `esimSevMessage` is called.

WARNING
: Idem for a warning.

ERROR Idem for an error.

FATAL Idem for a fatal message.

STATE_ENTRY
: This event is raised when the state is first entered.

STATE_EXIT
: This event is raised when the state is exited.

REAL_TIME_ERROR
: This event is raised in case of a real-time error.

REAL_TIME_MODE_ENTRY
: This event is raised at the transition to real-time mode, and at STATE_ENTRY when in real-time mode.

NON_REAL_TIME_MODE_ENTRY
: This event is raised at the transition to non real-time mode, and at STATE_ENTRY when in non real-time mode.

SNAPSHOT_END
: This event is raised after loading a snapshot and applying the values to the variables. Restoring a snapshot is performed asynchronous. This means that when the user issues the command, the snapshot is not applied when the command finishes. Instead this event is raised to indicate that it has finished.

11.3.4.3 Predefined output events

The following output events are predefined:

INIT Requests transition from 'Unconfigured' to the 'Initializing' state.

GO Requests transition from 'Standby' to the 'Executing' state.

RESET System reset. Requests transition from 'Standby' to the 'Initializing' state.

PAUSE Requests transition from 'Executing' to the 'Standby' state.

ABORT System abort. Requests transition from 'Standby' or 'Executing' to the 'Unconfigured' state.

STOP Request transition from 'Standby' to the 'Exiting' state.

QUIT Requests transition from 'Exiting' to the 'Unconfigured' state.

REAL_TIME_MODE
Requests transition to the real-time mode.

NON_REAL_TIME_MODE
Requests transition to the non real-time mode.

11.3.5 Tools menu

Schedule Configuration...
This menu item will show the Schedule Configuration dialog (see Figure 11.8).

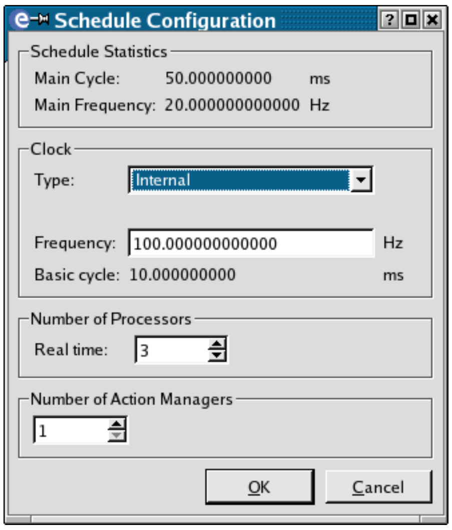

Schedule Configuration
Schedule Statistics
Main Cycle: 50.000000000 ms
Main Frequency: 20.000000000000 Hz
Clock
Type: Internal
Frequency: 100.000000000000 Hz
Basic cycle: 10.000000000 ms
Number of Processors
Real time: 3
Number of Action Managers
1

Figure 11.8: Schedule Configuration Dialog

In this dialog, the following properties of the schedule can be set:

Type This determines which clock is used by the scheduler. The availability of clocks depends on the selected model and target platform (see Section 11.4.9).

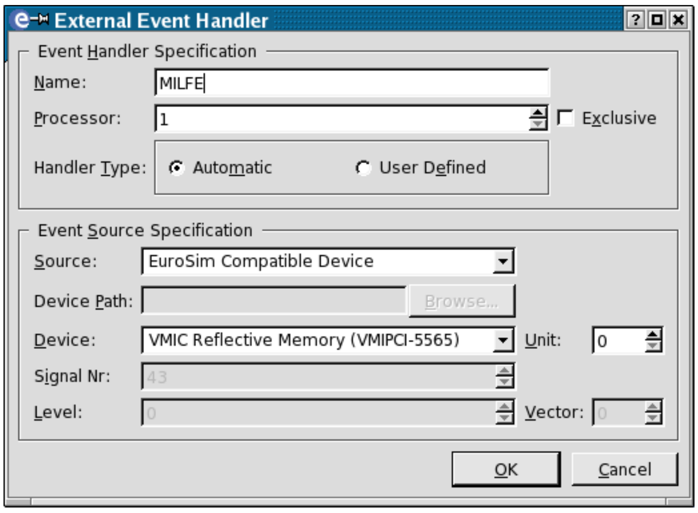
Period / Frequency
The desired period or frequency at which the scheduler should operate. The default is 100 Hz, but this can be raised up to 1000 Hz, depending on the clock type. The requested frequency is converted to a period in milliseconds. This period is used as the basis to calculate simulation time, so round numbers are in favour. Note that on some platforms it is possible to specify external clock sources. In that case it is important that you specify the right frequency for correct simulation time calculation.

Real time
The number of processors to be allocated to the scheduler. The maximum number of real-time processors is 10. The default value is 3 processors.

Number of Action Managers
The number of action managers which can be explicitly scheduled in each simulator state. The default value is 1.

External Event Handlers...
This menu item will show the list of External Event Handlers (see Figure 11.9). Here External Event Handlers can be added, deleted or modified. The user has to specify the processor

that handles the external event. With 'exclusive' use of the specified processor, the scheduler excludes the processor from the 'any' pool for task execution[1]. Event handlers that have an 'automatic' handler type, automatically add an input connector to the *Insert:External event* menu (see Section 11.3.4.1). The external event gets the same name as the event handler. Event handlers of handler type 'user defined', need additional code to handle the event and optionally raise one or more user defined input connectors, see Section 16.5.1.

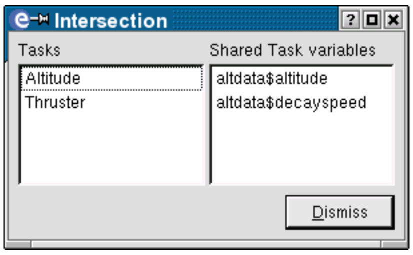


Name: MILFE
Processor: 1 [ ] Exclusive
Handler Type: [✓ Automatic] User Defined
Source: EuroSim Compatible Device
Device Path:
Device: VMIC Reflective Memory (VMIPCI-5565) Unit: 0
Signal Nr: 43
Level: 0 Vector: 0

Figure 11.9: External Event Handler Dialog

Intersection...

This item will show the Intersection dialog (see Figure 11.10). The Intersection window shows all variables that are shared by all the selected tasks. This way, it is easy to see if there are any (possibly unwanted) interactions between tasks.



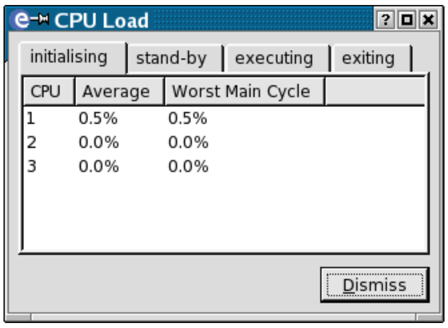
Tasks	Shared Task variables
Altitude	altdata$altitude
Thruster	altdata$decayspeed

Figure 11.10: Intersection Dialog

CPU *load...*

The fields of this window show the processor load for each of the processors per state of the schedule (see Figure 11.11). The processor load is calculated using the *mean* duration (execution) fields of the tasks. Timings for tasks assigned to 'Any' processor are split among all processors. If any of the processors has a load of more than 50%, this will result in a non-feasible schedule.

[1]This setting has no meaning on single CPU machines.

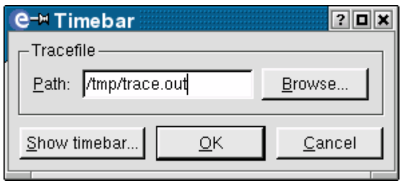


CPU	Average	Worst Main Cycle
1	0.5%	0.5%
2	0.0%	0.0%
3	0.0%	0.0%

Figure 11.11: CPU Load Dialog

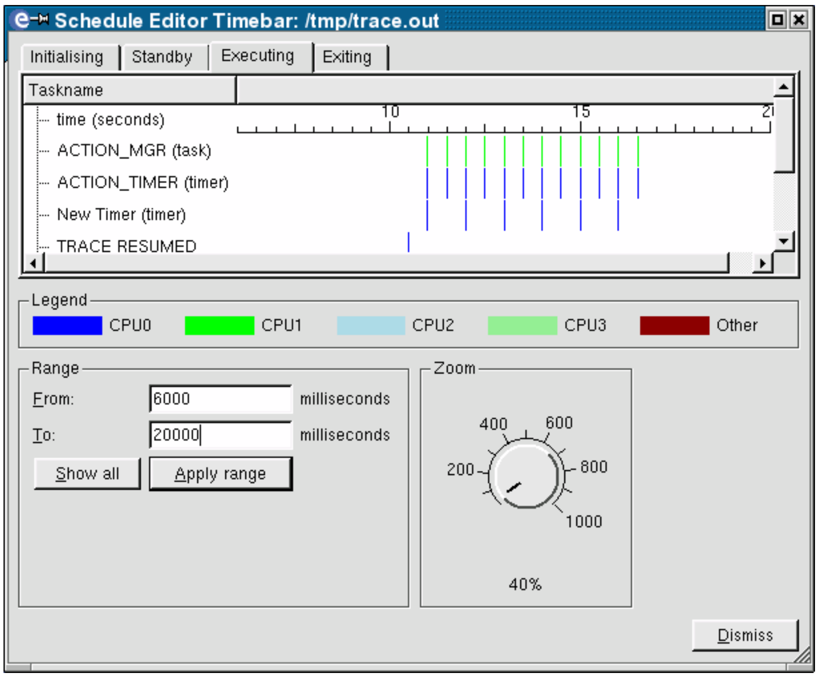
Timebar. . .

With the timebar dialog the scheduler trace file can be specified (see Figure 11.12) and viewed (see Figure 11.13). Note that the trace file can grow substantially, so only use it when you are debugging your schedule. Specify its location (path) somewhere on a local drive, thus avoid using a networked drive.



Path: /tmp/trace.out

Figure 11.12: Timebar Dialog



From: 6000 milliseconds

To: 20000 milliseconds

Figure 11.13: Timebar View

11.4 Advanced Scheduler topics

In this section some examples are given that will give more information on mutual exclusion behavior, the activation of user tasks according to mutual exclusions, dependencies, performing I/O in the non-real time domain, time requirements, how the scheduler will handle state transitions between different simulation states, and how to schedule the ActionMgr.

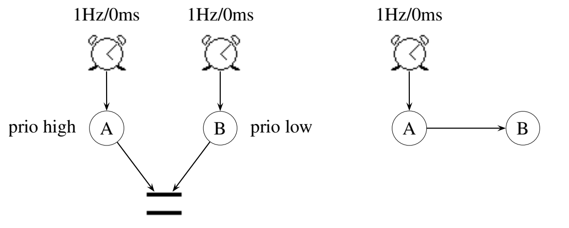
11.4.1 Scheduler mutual exclusion behavior

11.4.1.1 Effect of mutual exclusions

A mutual exclusion, or asynchronous store, in the Schedule Editor represents a 'mutual exclusive' run-time behavior between tasks. The task that captures the store first is allowed to continue running while all other tasks that are attached to that store, are prevented from starting until the store becomes available again (only one task can capture the store at any one time).

11.4.1.2 Effect of task priorities

Using priorities on tasks implies that when the task with the lowest priority is running and a task with a higher priority is activated, the task with the highest priority will preempt the lower priority task when that lower task is preemptable and no other processor is available.
Thus in the case that two tasks are connected to a mutual exclusion, using a higher priority for a task does not imply that that task will capture the mutual exclusion first, as it is the starting time that is of importance. If such a dependency is required, then it can be better specified using the following construction:

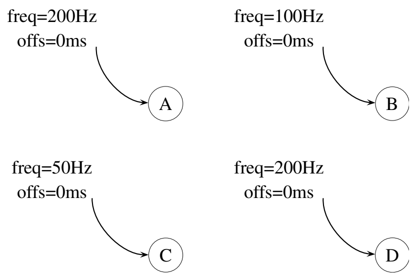


Wrong approach *Correct approach*

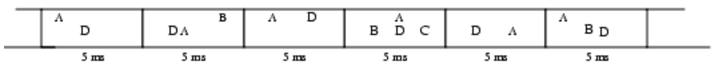
Note that even in the example above the starting time is never exactly the same, one of A or B will start slightly earlier than the other (the difference might be in nanoseconds). Which one in this case runs first depends on system internal behavior.

11.4.2 Dependencies, stores and frequency changers

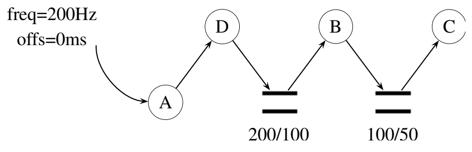
Dependencies, stores and frequency changers are used to define a sequence of tasks. Suppose that we have the following schedule:



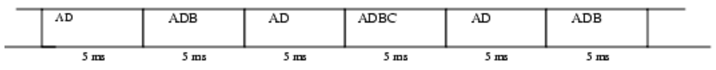
With this schedule it is defined that task A and D must be activated each 5 ms, task B must be activated each 10 ms, and task C must be activated each 20 ms. The maximum frequency on which the scheduler can activate tasks is for all states default 200 Hz. This means that the "real-time" is split up in time slots of 5 ms. For the example, the scheduler will activate tasks A and D in slot 1,2,3,..., task B in slot 2,4,6,..., and task C in slot 4,8,12,...



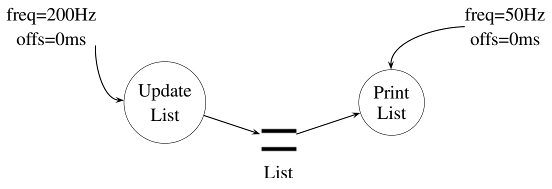
In the previous example, the sequence of tasks within the slots, is not defined. To define the sequence between tasks within the slots, dependencies (between tasks with the same frequency) and frequency changers (for tasks with different frequencies) can be used. In the following example the sequence of tasks within the time slots is defined with dependencies and frequency changers.



Note that the frequency of task D is still 200Hz, the frequency of task B is still 100Hz and the frequency of task C still 50Hz. These frequencies are now defined in the output frequency of the frequency changer. With these frequency changers it is defined that the time slots and sequences of tasks, within these slots, will be:

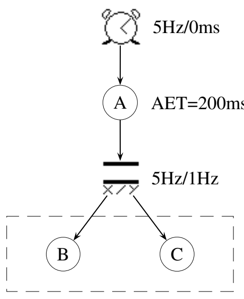


In the previous example we used frequency changers to define the sequences of tasks. With the defined sequence it is implicitly defined that tasks do not run simultaneous. If we do not want to define a sequence, but we only want to define that tasks are not executing simultaneous, we can use mutual exclusions. Tasks that read or write from the same mutual exclusion, are never executed by the scheduler simultaneous. For example, if we have a "printing" task that prints the contents of a linked list on 50 Hz, and a "updating" task that is changing the list at 200 Hz. It is obvious that the updating task may not run simultaneous with the printing task. To solve this problem, we can use a frequency changer.



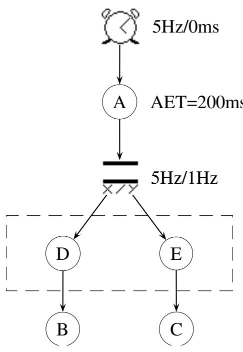
11.4.3 Frequency changers and mutual exclusive execution of tasks

The frequency changer takes care of mutual exclusive execution of the tasks that write to it with the tasks that read from it. In case of a N:1 frequency store, this can severely limit the allowed execution time of the reading tasks. This is explained using the drawing below:



In this figure, the frequency changer must guarantee that task A will run mutual exclusive with tasks B and C. The allowed execution time of task A is limited to a maximum of 200 msec as a consequence of the frequency of 5Hz.
After 5 activations of Sync Store, the store will activate tasks B and C, *before* releasing task A for the next activation. However, task A must be released in 200msec (its AET), or else it will cause real-time

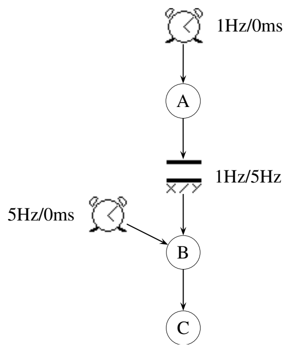
errors. The total allowed execution time of the combination of task B and task C is therefore limited to a maximum of 200msec. In practice, the duration of task A will be larger than zero, which further reduces the allowed execution time of B+C.
If the execution of B+C is more than allowed, a solution might be to store the part of the code that needs the mutual exclusive behavior in a separate task. For instance:



The part of the code of B and C that needs to be executed mutually exclusive with A (because it accesses the same variables) is stored in D and E. The remaining code is still in tasks B and C.
Now only the code in D and E must have a combined duration that is smaller than 200msec.
Note: D and E do not run mutually exclusive. If that is required, this can be accomplished by connecting these two tasks to a mutual exclusion (see Section 11.3), or even simpler by combining the code contained in D and C in one task.

11.4.4 Timing the output frequency of a frequency changer

Although a frequency changer has an output frequency, tasks reading from a frequency changer will only be activated with a frequency that approximates the specified output frequency. If more accuracy is desired, the frequency of the activations can be made exactly the one specified in the output frequency of the frequency changer by adding a timer. This is explained in the figure below:



Without the 5Hz timer, B is activated 5 times in rapid succession after each activation of A. Therefore the frequency of B would not be exactly 5 Hz, but would be determined by the execution duration of B. This is sufficient if only the ratio between A and B is of importance. However if it is required that B must be executed with an exact frequency of 5Hz, then the 5Hz timer should be added, which forces B to wait 200msec between the successive executions of B. The advantage of not adding a timer is that execution time is more efficiently used.

11.4.5 Example of using an output connector for I/O

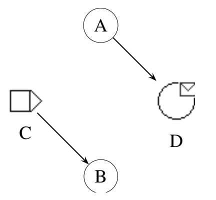
I/O is non-deterministic in time and thus calls must be issued from the non-real-time domain. In the Schedule Editor this can be achieved by connecting the task that performs the I/O to an output-connector. There are two ways to synchronize your non-real-time tasks with the real-time tasks:

1. You can synchronize explicitly in the Schedule Editor, using the schedule items available
2. You can use a ‘flag’ variable in memory to pass the status information about the I/O.

Both are explained below:

11.4.5.1 Using Schedule Editor items for synchronization

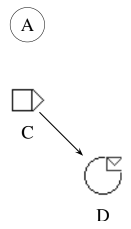
The following figure explains the first approach.



Task A performs some action. When finished, the non real-time task D is activated which performs the task D containing entrypoints that do the I/O actions. Within task D, when it has performed its I/O actions, a call to the function `esimRaiseEvent` is made (in this case with argument “C”). This function call activates the Input Connector C which in turn will activate Task Item B. Data read by task D can now be used by task B.

11.4.5.2 Using a variable for synchronization

Approach 1 implies that D is activated each time A was activated. Using a synchronous store a relation can be established (like for every N times A was activated D is activated once). You may want a more parallel behavior where tasks A and D run in parallel, and A uses the data read by D when available. This is described below:

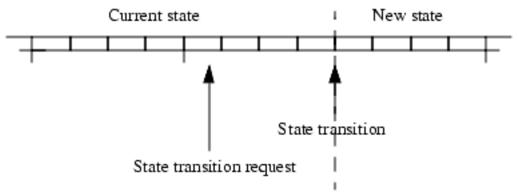


When task A needs to perform I/O, it sets a variable (e.g. `io_request`) and activates the input connector C by calling `esimRaiseEvent(C)`. Task A keeps on running.
The activation of C will cause an activation of D. Task D connected to non real-time task D will perform its I/O and will set a variable (for instance `io_handled`) when the I/O operation is ready.
While running, task A scans variable `io_handled` to verify if I/O has completed. When it detects that this variable has been set, both `io` variables can be reset, and data read in the I/O action can be used.
Note that, within this approach, it is also possible to activate input-connector C from a MDL script instead of a task. Using this feature, D can be activated from the Simulation Controller.

11.4.6 State transitions

A state transition can only occur at a main cycle boundary. A main cycle has a period equal to the least common multiple (LCM) of the periodic tasks computed over all states of the simulator in the schedule. In the current implementation, the main cycle is taken as the LCM of the periods of all periodic tasks (over all states), instead of the LCM of the periods of active tasks in the current running state. This, for reasons of simplicity, is still correct, although it may make the main cycle somewhat larger than strictly necessary.

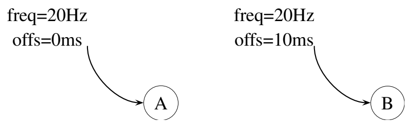
In the previous example we had a main cycle “AD:ADB:AD:ADBC”of 20 ms duration. This means that state transitions can only occur at each “4 slots” boundary. For this reason the scheduler will delay the user’s state transition request until the end of slot 4, 8, 12, . . . etc.



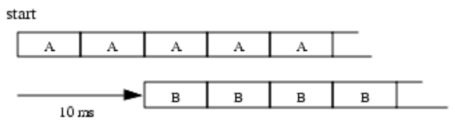
NB. If in the period between the request and the transition more state requests are given, these requests are buffered by the scheduler (up to 32) and applied on FIFO basis at the next main cycle boundaries, with one at a time.

11.4.7 Offsets

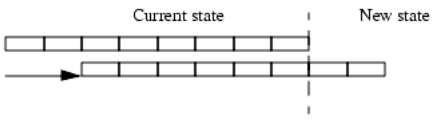
Offsets are used to “delay” tasks to following time slots. Suppose we have the following schedule:



The 10 ms offset of timer B will delay all activations of task B by 10 ms.



When offsets are used, state transitions will still be on the main cycle boundaries. This means that task B must still be activated (according to the current executing schedule), in the first two slots of the new state. This guarantees that the number of activations for each tasks are always the same. I.e. a functional model will always complete leaving the system in a deterministic state.

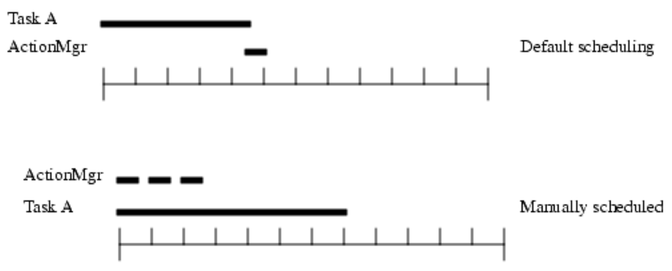


Note that no synchronization whatsoever is performed between the schedules in the ‘old’ and ‘new’ state: this is omitted under the assumption that there is only one nontrivial EuroSim state (state EXECUTING), and that any other state is to perform simple procedures, such as initialization or keeping hardware alive. Supporting state synchronization would unnecessarily add to the complexity of the scheduler. The user must however be aware of a possible overlap in execution of the schedules of two states ‘just after’ a state transition when offsets are used.
Note: One exception is made for the transition to ABORT. An abort transition does not wait until the main cycle boundary, but is directly done by the scheduler. This means that all tasks, inclusive tasks with an offset, are directly stopped.

11.4.8 Scheduling the action manager (ACTION_MGR)

The action manager is a special task provided by the EuroSim environment. Although it is a special task, the action manager must be scheduled just as any normal task. As with any normal task, how it

is scheduled is of importance to its performance. For instance, if variables are to be logged just after performing a certain task, then the action manager could best be scheduled after this task using a flow (dependency relation).
When the action manager is not scheduled explicitly, i.e. not placed on the tab page in the Schedule Editor, the action manager is added to the schedule with a default frequency that is equal to the Basic Frequency of the scheduler and with a priority of Low. In many cases this will be sufficient, as this will activate the action manager with a high frequency, and after all other tasks have been activated.
However, there are cases where the action manager should be scheduled more carefully using the Schedule Editor. One such case has already been mentioned: to provide logging of variables on a specific moment in the overall schedule. Another example is the case in which only one real-time executor is available on which a low frequency task with long duration is running. Due to its long duration some time slots are filled completely, leaving no time to run the action manager. In this case the default Low priority will lead to real-time errors. Scheduling the action manager in the Schedule Editor with a higher priority may be the solution. This is illustrated below:



Default vs Manual scheduling of the ActionMgr, when having a long-duration task

11.4.8.1 Multiple action managers

There are situations where a single action manager does not allow you to execute the actions at the appropriate place in the schedule. For that situation it is possible to specify more than one action manager task. The number of action managers can be configured in the Schedule Configuration dialog box (see Section 11.3.5).
Each action manager can be scheduled individually at different frequencies in each scheduler state.
When there is only a single action manager it has the name ACTION_MGR. In the case when there is more than one action manager, the names are ACTION_MGR_0, ACTION_MGR_1, etc. The number corresponds to the action manager number you can specify for each individual action in the script dialog box in the Simulation Controller (see Section 12.13.1).

11.4.9 Clock types

Depending on the platform the simulator will be running on, the developer can choose from a number of clock types (or clock 'sources'). The type of clock to be used by the scheduler can be configured in the Schedule Editor (see Section 11.3.5). Note that for all external clock sources it is important that you specify the right frequency/period for correct simulation time calculation.
The following clock types are available on all platforms:

Internal
Represents the internal clock of the computer running the simulation.

As Fast As Possible
Runs the simulation as fast as possible (see Appendix K).

The following clock types are available on Irix and Linux:

IRIG-B The Datum IRIG-B card (bc635PCI).

Signal Wait for the specified signal number to be raised.

EuroSim Compatible Device
Currently the following devices are supported: The SBS PCI-VME card (Model 616/617) and the VMIC Reflective Memory card (VMIPCI-5565). When a device driver is used that meets the EuroSim interface requirements, then that device can also be used as a clock source. For more information on customization contact your EuroSim representative.

The following clock type is only available on Irix:

External Interrupts
Uses the SGI external interrupt facility.

Chapter 12

Simulation Controller reference

This chapter provides details on the Simulation Controller. The panes and tab pages of the editor, the various objects that can be created, all menu items of the editor and their options are described. For menu items not described in this chapter, refer to Section 3.5.

12.1 Starting the Simulation Controller

The Simulation Controller can be started by selecting the **Simulation Controller** button in the EuroSim start-up window (see Section 5.2.3), by selecting the **Observer** button in the start-up window, or via the command line.
When the Simulation Controller is started from the command line, the user can provide the following command line options:

-observer
Start the simulation controller in observer mode

-connect hostname:prefcon
Connect at start-up to an already running simulator running on host *hostname* on connection *prefcon*.

See also the manual page for the Simulation Controller *SimulationCtrl(1)*.
Example:

```
hobbes:~$ SimulationCtrl -connect minbar:0
```

Before components for a new scenario can be defined in the Simulation Controller editor, a model and a schedule should be selected. The model is needed for the definition of the scenario actions and the initial condition files using the data dictionary specific for that model. The schedule is required in order to actually run a simulation. By selecting the *File:New* menu item a wizard will appear that helps you select the files you need.
If the Simulation Controller is started by selecting the **Observer** button, then the number of options will be limited, as the outcome of the test cannot be affected in any way. This means that some menu options (e.g. debugging) and some activities (e.g. using a script to update a data value) are not available.
Before a simulation can be started through the Simulation Controller, a simulation definition file has to be loaded (using the normal *File:Open* menu item), or should be created (using the normal *File:New* menu item).

12.1.1 Converting EuroSim Mk2 missions

If you want to convert an existing EuroSim Mk2 mission (`.mdl` file), then you can use *File:Open* to select the existing `mdl` file (first choose *Mdl files* in the *File Type* combobox). Based on this mission file a new Simulation Definition file will be created and also `.usr` files if needed.
Please note that when you save the new Simulation Definition the original mission file will be overwritten and the original information in the mission file concerning Initial Condition files and the User Program

definitions will be lost. This information is now stored in the Simulation Definition file and in the `.usr` files.

12.2 Simulation Controller Input Files

The Simulation Controller allows the Test Conductor to create different simulation definitions for executing a model in the simulator, each testing e.g. a particular aspect of the model. Such a definition consists of the following components:

Reference to a model
: This is a link to a model definition. This link is necessary to collect all required information about a model.

Reference to a schedule
: This is a link to a schedule definition. This link is necessary to actually run a simulation.

Reference to an export
: This is a link to an export definition. This link is optional and specifies the exports file that describes which variable nodes will be exported to external clients, see file formats in Appendix F for a description on the exports file format. Chapter 18 describes in more detail how an exports file is used.

Initial conditions
: These are used to change the initial state of the model. The initial conditions override the initial values of the variables defined in the code.

Scenarios
: These are used to create events and actions, e.g. to introduce malfunctions in the simulation. A scenario contains script, recorder and stimulus actions. Several scenarios can be loaded at one time.

Stimuli files
: Stimuli are used to replace external data inputs which would be present in the real world. Time-series stimuli have their values taken from a file, for example to feed in values representing an operator's input. Functional stimuli have their values generated from a mathematical function.

MMI *Definitions*
: MMI definitions describe where monitors are placed on the MMI tab page and which data they monitor. Monitors on an active MMI page collect data during a simulation run. They do not store the information in a file, but display the data directly on screen. It is also possible to execute scenario scripts and activate/deactivate recorders and stimulus actions by placing buttons or checkboxes on the MMI tab page. In order to reduce required bandwidth between the simulator and Simulation Controller, you can deactivate an MMI file.

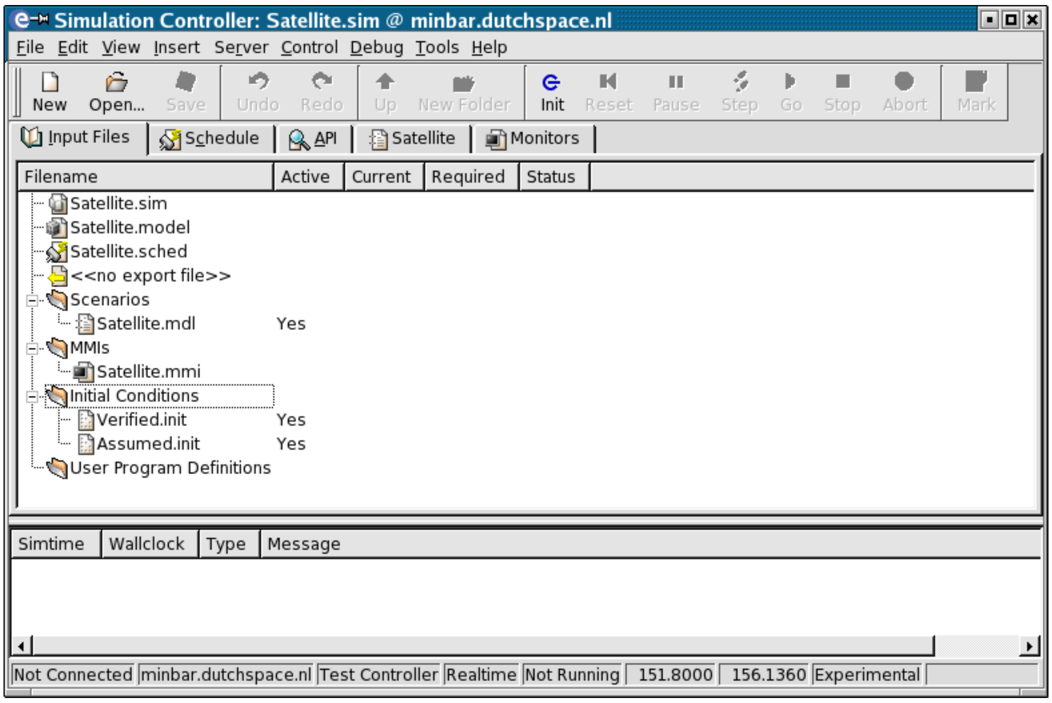
Image Definitions
: The simulation definition can contain information about one or more image definitions. Once the simulation has been initialized, an image definition can be "launched" as a separate client.

User Program Definitions
: A user program definition is used to launch a program as a separate client. That program can connect to the simulator and provide additional functionality.

Monitors could be stored in a scenario, but this is obsolescent. Instead, monitors should be defined in MMI files. If an old scenario containing monitors is imported, then the monitors should be converted using the menu *Tools:Convert Old Monitors*. Existing monitors in a scenario cannot be edited, only converted and deleted.
Not all of these components have to be present in one simulation definition. Only the references to the model and schedule are required.

12.2.1 Initial Condition

A particular simulation is often required to be executed several times, each one starting from a different state i.e. a different initial condition definition. Instead of creating different simulation definitions for each of these possibilities, it is easier to reference all the possible initial conditions within a single simulation definition, and then to ensure that the required initial conditions are selected prior to initializing the simulator.


Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API Satellite Monitors

Filename	Active	Current	Required	Status
Satellite.sim				
Satellite.model				
Satellite.sched				
<<no export file>>				
Scenarios				
Satellite.mdl	Yes			
MMIs				
Satellite.mmi				
Initial Conditions				
Verified.init	Yes			
Assumed.init	Yes			
User Program Definitions				

Simtime	Wallclock	Type	Message

Not Connected | minbar.dutchspace.nl | Test Controller | Realtime | Not Running | 151.8000 | 156.1360 | Experimental

Figure 12.1: Simulation Controller with multiple Initial Conditions

The required (active) initial conditions are indicated in the Input Files tab page: the initial conditions marked *Active* form the set of values that will be applied if you request “Init” or “Reset” from the Simulation Controller. Values which have been updated are then used in tasks scheduled for the “initializing” state. The set of active initial conditions can be updated by activating or deactivating the appropriate file in the Input Files tab page.
Alternatively, you can request *Control:Apply Initial Condition…* from the Simulation Controller to cause the data values within the file to be applied directly to the current simulation. In this case, the values are used to override the current simulation values. The simulation state is not affected when this option is used.

12.2.2 Script Action

This type of action contains a Mission Definition Language (MDL) script. A script is the basic building block from which all actions can be made. For ease of use, EuroSim provides special-purpose interfaces for recorders and stimuli. However, any actions which require more complex activation conditions (e.g. a recorder which is to record when a particular data value is between predefined boundaries) can only be made by defining the script directly.
MDL is a simple yet versatile language for simulation scripting. It allows users to write control scripts in a limited free-text, C-like language. Appendix E contains a comprehensive overview of MDL.
A script action is made up from four parts:

name Used to reference the action.

attributes
Which determine how the action looks on the scenario tab page, in which state it should be executed, etc.

execution condition
Which contains the condition (written in MDL) under which the action will be executed.

action to be executed
Which contains the actual MDL script which will be executed when the condition is true.

All of these items can be modified with the Action Editor, which is described in more detail in Section 12.13. The Action Editor is started when creating a new action, or when modifying an existing action.

12.2.3 Stimulus Action

The stimulus action is a special case of the script action, and can be used to easily create actions that provide stimuli to the simulator, using data from a specified file to update the values of the selected variables, at a certain frequency and for a certain time period. Using the *Variables* tab page in the Action Editor, there is no need for the user to write the MDL script himself. However, if needed, users can still access the raw MDL script, allowing the editor to be used for the creation of the basic stimulus action and then be customized.
See Section 12.13.3 for a more detailed description of the stimulus Action Editor.

12.2.4 Recorder Action

The recorder action is also a special case of the script action, and can be used to easily create actions that record the values of one or more selected variables, at a certain frequency and for a certain time period. Using the *Variables* tab page in the Action Editor, there is no need for the user to write the MDL script himself. However, if needed, users can still access the raw MDL script, allowing this editor to be used for the creation of the basic recorder action, and then be customized.
See Section 12.13.2 for a more detailed description of the recorder Action Editor.

12.2.5 Monitors

While it is possible to create a monitor script action, this type of monitor has become obsolescent. Generally you only come across a monitor action when loading an old (EuroSim Mk2 or earlier) `.mdl` scenario file or when you explicitly create a script action containing a monitor.
When an obsolescent monitor action is triggered a new tab page *Script Monitors* will appear that contains the created monitor.
In EuroSim Mk4.0 a monitor is no longer a script action. Instead monitors are defined in a `.mmi` file and can be edited in the corresponding MMI tab page. You can create multiple MMI tab pages, each containing a set of monitors.
In order to reduce required bandwidth between the simulator and Simulation Controller, you can deactivate an MMI file. When and MMI file is inactive, its monitors will not be subscribed for updates from the simulator. You can activate or deactive an MMI file when the simulator is running. The monitors will then subscribe or unsubscribe for updates as appropriate.
Monitors on the scenario tab page can be converted to an MMI tab page by using *Tools:Convert Old Monitors*.
There are two types of monitors: alpha-numerical and graphical monitors.
With alpha-numeric monitors, a window will be shown in the MMI tab page in which the current value of one or more variables will be presented. The window will be updated when the value changes.
Graphical monitors use one of three types of graphs to display the values of variables:

XY Plot one or more variables against an independent variable.

Simulation Time
Plot one or more variables against the simulation time.

Wall Clock Time
Plot one or more variables against the wall clock time.

See Section 12.14.3 for a more detailed description of the Monitor Editor.

12.3 Simulation Controller windows

When the Simulation Controller has been started, a window similar to the one in Section 12.3.3 is shown. This window is divided into two main parts, separated by a splitter:

Tab pane
: This pane contains several tab pages that used for editing, debugging and viewing a simulation.

Message pane
: Shows the messages from the simulator.

At the top is the menu bar and a tool bar. At the bottom a status bar provides additional state information.

12.3.1 The toolbar

The tool bar provides easy access to the following functions:

New Create a new Simulation Definition. The same as the *File:New* menu item.

Open
Open an existing Simulation Definition. The same as the *File:Open* menu item.

Save Save the current Simulation Definition. The same as the *File:Save* menu item.

Up Go up one level in the folder hierarchy. Available when the scenario is represented using icons. The same as the *View:Up* menu item.

New Folder
Create a new folder. Available in the scenario tab page. The same as the *Insert:New Folder* menu item.

Init Initialize the simulator. The same as the *Control:Init* menu item.

Reset
Reset the simulation. The same as the *Control:Reset* menu item.

Pause
Pause the simulation. The same as the *Control:Pause* menu item.

Step Advance the simulation through one executing cycle. The same as the *Control:Step* menu item.

Go Put the simulation in executing state. The same as the *Control:Go* menu item.

Stop Stop the simulation. The same as the *Control:Stop* menu item.

Abort
Abort the simulation. The same as the *Control:Abort* menu item.

Mark
Place a mark in the journal file. The same as the *Insert:Mark Journal* menu item.

12.3.2 The tab pane

The tab pane consists of the following tab pages:

Input Files
: Shows all files used by the Simulation Definition.

Schedule
: Used to debug a simulation run.

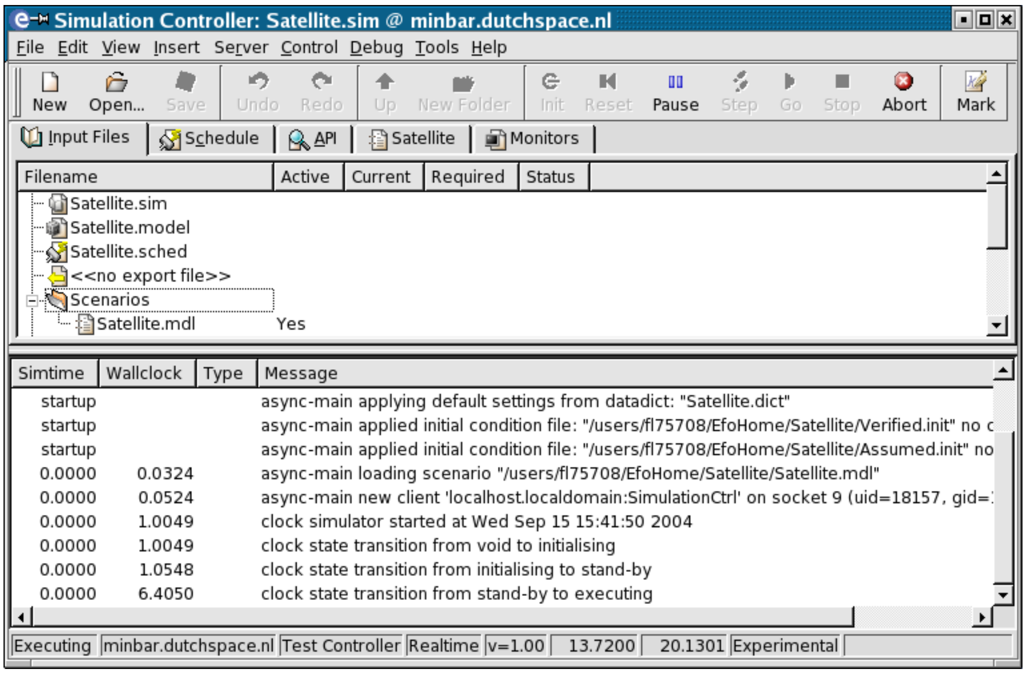
API Show the data dictionary and quickly monitor and/or change the value of a variable.

Scenario
: View and edit all actions in a scenario. One tab page appears for each scenario in the Simulation Definition.

MMI
: The Man-Machine Interface. One tab page appears for each MMI file in the Simulation Definition. The MMI tab page allows you to monitor variables and to execute scripts, recorders or stimuli.

12.3.3 The message pane

On the message pane all messages are displayed. This includes messages generated by the simulator (e.g. when starting the simulator, or when pausing it), errors from the scheduler (see Appendix C). as well as marks and comments created by the test conductor. Comments are marks with an extra item of text attached. See Section 12.3.3 for some examples. Marks and comments can be created with the *Insert:Mark Journal* and *Insert:Comment Journal Mark* menu items. All messages appearing on the pane are also logged into the journal file, see Section 12.4.


Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API Satellite Monitors

```
Filename              Active  Current  Required  Status
Satellite.sim
Satellite.model
Satellite.sched
<<no export file>>
Scenarios
  Satellite.mdl       Yes
```

```
Simtime   Wallclock  Type  Message
startup                    async-main applying default settings from datadict: "Satellite.dict"
startup                    async-main applied initial condition file: "/users/fl75708/EfoHome/Satellite/Verified.init" no c
startup                    async-main applied initial condition file: "/users/fl75708/EfoHome/Satellite/Assumed.init" no
0.0000    0.0324           async-main loading scenario "/users/fl75708/EfoHome/Satellite/Satellite.mdl"
0.0000    0.0524           async-main new client 'localhost.localdomain:SimulationCtrl' on socket 9 (uid=18157, gid=:
0.0000    1.0049           clock simulator started at Wed Sep 15 15:41:50 2004
0.0000    1.0049           clock state transition from void to initialising
0.0000    1.0548           clock state transition from initialising to stand-by
0.0000    6.4050           clock state transition from stand-by to executing
```

```
Executing | minbar.dutchspace.nl | Test Controller | Realtime | v=1.00 | 13.7200 | 20.1301 | Experimental
```

Figure 12.2: The Simulation Controller

Messages generated by the simulator include messages about:

- Change of state.
- Problems encountered, such as real-time errors.
- Manual activation of actions.
- Updates to the action definitions

12.3.4 The status bar

In the status bar a number of items about the current simulation are displayed:

- The current simulation state.
- The simulation server.
- The current user role (Test Conductor or Observer)

- The simulation mode (real-time vs. non-real-time vs. debug)
- The simulation speed.
- The simulation time (it is expressed in seconds or as an absolute time displayed as YYYY-mm-dd HH:MM:SS.ssss if the simulation uses UTC).
- The wall clock time (elapsed time since start-up or the UTC time if the simulation uses UTC).
- Traceability: experimental or traceable. If the simulation of a versioned simulation definition is requested, then various checks will be carried out to assess whether the execution will be traceable at a later date or not. If so, then the status bar will state that the simulation is *Traceable*, if not, then the simulation is *Experimental*.

'Traceability' means that all source files involved in the simulation definition can themselves be traced at a later date. This is only possible if a) the source files (i.e. simulation definition, scenarios, initial conditions, executable, MMI files, data dictionary and schedule (the latter deriving from the model file)) are (generated from) non-modified repository versions (e.g. `1.2` not `1.2+`) and b) the versions on disk match the required versions.

12.4 Simulation Controller output files

During a simulation run, a number of files are generated:

journal file
: This file contains all messages generated by the simulator, as well as all entered marks and comments. There are two variants of this file. A human readable version and a machine readable version. The filename of the human readable file is `EsimJournal.txt`. The filename of the machine readable file is `EsimJournal.xml`.

timings file
: This file contains timing information which can be used in a schedule (see Section 11.3.1 of the Schedule Editor). This file has the name `timings`. See also Section 11.4 for information on task timings.

recording files
: These are the files that result from the recording actions as defined in the scenario definition. For each recorder a file is created with the name *recordername*`.rec` if the default name was chosen in the scenario definition.

test result file
: This file contains a list of all recordings performed during the simulation run. This file will have the extension `.tr`.

All these files are created in a directory with a name like `2001-12-14/15:33:30`, which includes the date and time of the simulation run.

12.5 Dictionary Browser reference

The Dictionary Browser allows the Simulation Controller and other programs to look at which variables and entrypoints have been defined in the API headers of the model, and therefore are available in the data dictionary.
The browser shows a tree hierarchy of the available nodes, files, entrypoints and variables. If you try to expand a very large array, then you will be asked for a confirmation first. The selected items can be dragged and dropped to the destination. Double clicking on a single item will also add that variable to the destination. There is also a button **Add** to add the selected variables to the destination.

You can switch between a full view and a condensed view where all unnecessary nodes are left out by pressing the *F3* key or by choosing *Condensed View* or *Full View* from the context menu that you get when pressing the right mouse button in the Dictionary Browser.
If you want to find a variable you can either choose *Find* from the context menu or start typing immediately while the Dictionary Browser has the input focus. For every key you type the browser will be updated to show only those variables that match the text you've typed. The browser uses a case-insensitive substring search. So any variable name that contains the text without regard to upper or lower case will match. When no variables match the browser is empty. Use backspace to delete the last character from the search string until the search string is empty, and then you return to the original state of the browser.
Note that the search string is also displayed in the caption of the first column of the dictionary browser.
The context menu also contains a *Expand All* item to expand all nodes and a *Collapse All* item to collapse all nodes in the tree.
Finally, there is a *Info* menu item in the context menu that appears when you click with the right mouse button on a node in the dictionary. Selecting this menu item will pop up a window that shows type information about the selected node.

12.6 Initial Condition Editor reference

The Initial Condition editor allows the specification of a particular state to which the model should be initialized prior to execution, e.g. locations of payloads or the state of hatches. It is only necessary to specify values in the initial conditions if these values override the initial value specified in the API header.
The initial conditions are set prior to execution of the code, and a simulation can be re-initialized during a run.
The validity of the initial condition cannot be checked by EuroSim. However, the Initial Condition editor will only allow values of the correct type to be entered which are the range that was specified in the API headers of the model.
The initialization sequence is as follows:

- first the simulator is loaded and the variables will get the values as they are hard coded in the source file.
- next the model is loaded and the variables defined in the API headers will get their designated default values
- finally, the initial conditions are used to set the variables specified in the Initial Condition files, with their values. The order of appearance in the *Input Files* tab page determines the order of initialization. I.e., the top-most Initial Condition file is applied first, followed by the second file, etc.

12.6.1 Starting the Initial Condition editor

The editor is started by double-clicking with the left mouse button on an Initial Condition file in the *Input Files* tab page, or by selecting an Initial Condition file and then selecting *Edit:Properties*. A dialog appears that uses the Dictionary Browser to represent the dictionary and to edit the initial conditions.
You can set initial values by left-clicking on the line containing the variable that you want to edit or by selecting the line and pressing F2.
Values that are out of bounds are rejected. If you want to set the initial value for a variable designated as a parameter then a window appears asking for confirmation.
You remove an initial value by clearing the contents. However, clearing a member of a structure or array will only reset the value to the default value. If you want to clear the initial value of the whole compound variable, then right click on the top variable node and select *Clear* from the context menu.
If the initial value that you entered is equal to the default value, then the initial value is cleared and removed from the set of initial condition values. As indicated above, this does not apply to the members of compound variables.

Any variable that has an initial value is marked with a small asterisk (*). Also all entrypoint and org nodes that contain variables that have an initial value are marked the same way.

12.6.2 Context menu items

If you right click on a node or on the background a context menu appears with the following items (besides the menu items that are described in Section 12.5):

Clear The initial value is removed for the selected variable.

Show Modifications Only/Show All
This menu item toggles between showing all variables or only those that have an initial value. You can also use the key *F4* as a shortcut.

Undo Undo the last change.

Redo Redo the last *Undo* action.

12.7 Simulation Controller Menu Items

This section describes the menu items that are not tied to a specific tab page and that do not belong to the group of common menu items that are described in Section 3.5.
Menu items that are only enabled when a specific tab page is on top are described in the section for that tab page.

12.7.1 View menu

Input Files
Raise the Input Files tab page to the top.

Schedule
Raise the Schedule tab page to the top.

API Raise the API tab page to the top.

Script Monitors
Raise the Script Monitors tab page to the top.

MMI A sub-menu with all MMI tab pages. The selected tab page will be raised to the top.

Scenarios
A sub-menu with all Scenario tab pages. The selected tab page will be raised to the top.

Toolbar Button Labels
Show text below the toolbar buttons. This setting is saved in a settings file and will be restored the next time the Simulation Controller is started.

Large Toolbar Buttons
Show large icons for the toolbar buttons instead of the default small icons. This setting is saved in a settings file and will be restored the next time the Simulation Controller is started.

Tabbar Labels
Show text on the tab-bar. Disabling this setting can be useful if your Simulation Definition file contains a lot of MMI and/or script files. This setting is saved in a settings file and will be restored the next time the Simulation Controller is started.

Refresh If the data dictionary or schedule file have been changed, then reload these files.

12.7.2 Insert menu

New Scenario
: Add a new Scenario file to the Simulation Definition. This will automatically create a new Scenario tab page where this file can be edited. You will be asked to enter the caption of the new tab page.

Add Scenario
: Import an existing scenario file into the Simulation Definition. A new tab page will be created where this file can be edited. You will be asked to enter the caption of the new tab page.

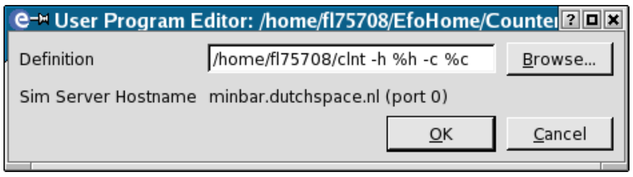
New MMI
: Add a new MMI file to the Simulation Definition. A new MMI tab page will appear where you can add monitors, etc. You will be asked to enter the caption of the new tab page. By default the new MMI file will be marked as *Active* in the Input Files tab page.

Add MMI
: Import an existing MMI file into the Simulation Definition. A new tab page will be created where this file can be edited. You will be asked to enter the caption of the new tab page. By default the imported MMI file will be marked as *Active* in the Input Files tab page.

New Initial Condition
: Add a new Initial Condition file to the Simulation Definition. By default the new initial condition file will be marked as *Active* in the Input Files tab page.

Add Initial Condition
: Import an existing Initial Condition file into the Simulation Definition. By default the imported initial condition file will be marked as *Active* in the Input Files tab page.

New User Program Definition
: Create a new User Program Definition. This is basically a user defined program that will be launched when you select *Edit:Launch*. The User Program Definition window is very simple (see Figure 12.3). In the *Definition* input field the program to start is specified and any arguments that are needed. The `%h` sequence will be replaced with the hostname of the running simulator, and the `%c` sequence will be replaced with the preferred connection number. If you need to run `.bat` batch files (Windows version only), then you have to precede the User Program Definition with `'cmd /C '`. Similarly for shell scripts (`.sh` files); precede the User Program Definition with `'bash '`. If the shell script file is located in the same directory as the `.sim` file and you do not specify the full path to it, then you may need to prefix the name of the shell script file with a `'./'`, depending on whether the current directory (dot) is in your search path or not (environment variable PATH). Examples: `'bash -c ./myscript.sh'` or `'cmd /C mybatch.bat'`.


User Program Editor: /home/fl75708/EfoHome/Counte

Definition /home/fl75708/clnt -h %h -c %c

Sim Server Hostname minbar.dutchspace.nl (port 0)

Figure 12.3: Example User Program Definition

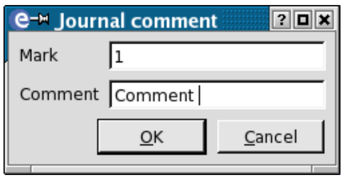
Add User Program Definition
: Import an existing User Program Definition.

Make Mark
: Use this menu item to make a mark in the simulation log. The mark is also displayed on the message pane. The idea behind marks is to allow you to tag some interesting/unexpected event quickly. Each mark is allocated a unique number which can also be used for adding explanatory comments later on.

Make Comment

Use this menu item to enter a comment in the simulation log. The comment is also shown on the message pane. When this menu item is selected, a window shown in Figure 12.4 will pop up, in which the comment can be entered.

By default, the comment 'belongs' to the last mark made, but you can add comments to earlier marks by manually editing the number in the Mark field.



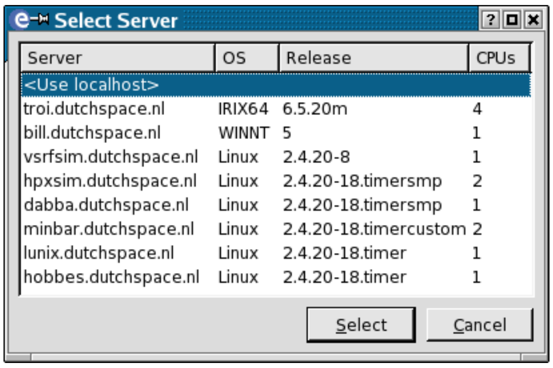
Mark	1
Comment	Comment

Figure 12.4: The Comment Journal Mark window

12.7.3 Server menu

Select Server

Before a simulation can be started, a computer on the network has to be selected which can act as the simulation server. By default the host on which you started EuroSim is assumed to be the simulation server, and so this option is only necessary if you wish to use another host. When this menu item is selected, a window similar to the one in Figure 12.5 is shown. This window lists all currently available servers on the network. Use the *Server:Show Current Simulations* menu item to check the status of each of those servers.



Server	OS	Release	CPUs
<Use localhost>			
troi.dutchspace.nl	IRIX64	6.5.20m	4
bill.dutchspace.nl	WINNT	5	1
vsrfsim.dutchspace.nl	Linux	2.4.20-8	1
hpxsim.dutchspace.nl	Linux	2.4.20-18.timersmp	2
dabba.dutchspace.nl	Linux	2.4.20-18.timersmp	1
minbar.dutchspace.nl	Linux	2.4.20-18.timercustom	2
lunix.dutchspace.nl	Linux	2.4.20-18.timer	1
hobbes.dutchspace.nl	Linux	2.4.20-18.timer	1

Figure 12.5: Select Server window

Show Current Simulations

Use this menu item to check the status of each of the available simulation servers with respect to the number of simulations running on those servers. An example is shown in Figure 12.6. The **Show Paths** button can be used to show the exact path of each the simulation running on the servers. When the paths are shown, the button will change into a **Hide Paths** button, which reverses the action. The **(Re)Connect** button can be used to connect to one of the simulation servers shown. The **Kill Sim** button can be used to kill a simulation if a run is hanging for any reason and is no longer responding to the Simulation Controller.

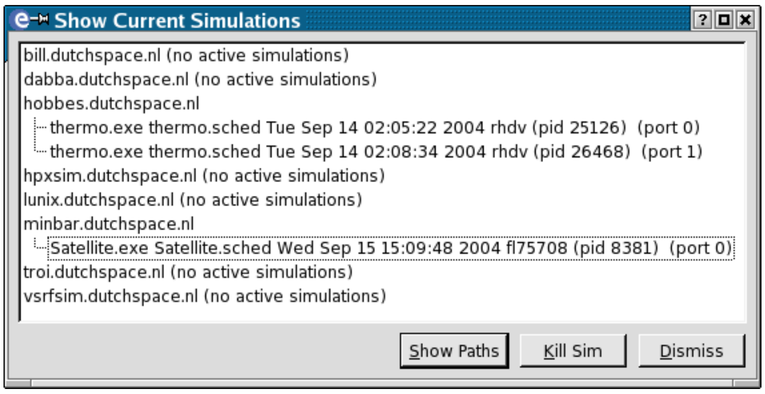

```
bill.dutchspace.nl (no active simulations)
dabba.dutchspace.nl (no active simulations)
hobbes.dutchspace.nl
  thermo.exe thermo.sched Tue Sep 14 02:05:22 2004 rhdv (pid 25126)  (port 0)
  thermo.exe thermo.sched Tue Sep 14 02:08:34 2004 rhdv (pid 26468)  (port 1)
hpxsim.dutchspace.nl (no active simulations)
lunix.dutchspace.nl (no active simulations)
minbar.dutchspace.nl
  Satellite.exe Satellite.sched Wed Sep 15 15:09:48 2004 fl75708 (pid 8381)  (port 0)
troi.dutchspace.nl (no active simulations)
vsrfsim.dutchspace.nl (no active simulations)
```

Figure 12.6: Show Current Simulations window

Disconnect From Server
This menu option will disconnect the Simulation Controller from the simulation server. The simulation will remain on the server, and the Simulation Controller can be reconnected to the server using the *Server:Show Current Simulations* menu item.

12.7.4 Control menu

Set Realtime
This menu item acts as a toggle with which the simulation can be set to real-time mode or non-real-time mode. This can only be done before initializing the simulator.

Speed Control
Use this menu item to get the Speed Control Window as shown in Figure 12.7. When the simulation is running non real time the user can speed up or slow down the scheduler clock with the slider. The 'as fast as possible' button selects a mode where the scheduler is boosted to maximum speed without internal clock overhead. The actual speed can be lower than the requested speed, since the scheduler slows down if tasks do not complete in time[1].

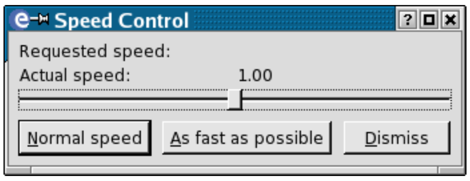

Requested speed:
Actual speed: 1.00

Figure 12.7: The Speed Control window

Init This will initialize the simulator. This process comprises of the following steps:

1. Load the application model associated with the current simulation definition.
2. Use the data dictionary information to set initial values.
3. Use the Initial Condition files (if active) to update initial values.
4. Execute the task from the initializing schedule through the scheduler.
5. Execute the actions that are tagged as active during the initializing state. Once the initialization is complete, the simulator will be in the standby state at simulation time 0.0000 seconds, or the simulation time set by a script or model code.

Reset This will reset the simulation (i.e. perform steps 2 through 5 of the initialization process). Note that if the schedule contains an output connector connected to ABORT, the simulation cannot be reset.

[1]Speed Control has no effect if an external clock is used whose frequency cannot be changed by EuroSim.

Step This will advance the simulation through one executing cycle. If the schedule contains a low frequency task, then this could be a significant period of time.

Go This will put the simulator in the executing state.

Pause This will temporarily stop the simulation (put it in standby state). The simulation is not necessarily completely inactive however, as tasks and actions specified for the standby state will be still executed.

Stop This will stop the simulation gracefully. The simulator will be transitioned to the exit state and all open files will be properly closed.

Abort This will abort the simulation instantaneously. Open files will not be closed by EuroSim, but rather by the operating system, which results in loss of data as data still in memory is not saved.

If a test execution has resulted in a simulator hang, or remaining executables from previous simulation runs, use the *Server:Show Current Simulations* menu option and select the offending simulation and request **Kill Sim** to remove the remaining executables.[2]

Raise Event
Show a list of available user defined events. Select an event and raise that event by either double clicking the event or pressing the **Raise Event** button. This menu item is only available when the connection to the simulator is active and if at least one user defined event is available.

Suspend/Resume Recording
This menu option allows the user to activate/deactivate all recording actions in the simulation via a single request. This can be useful for temporarily suspending recording during a simulation run.

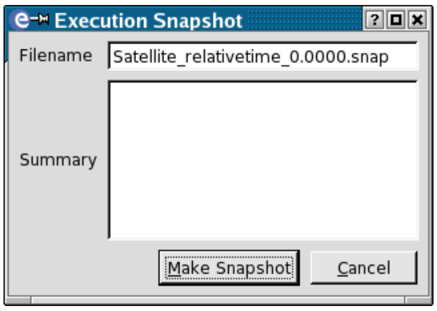


Figure 12.8: Take Snapshot window

Take Snapshot
This menu option will pop-up a window (see Figure 12.8) with which a snapshot of the current state of all simulation variables can be made. In the same window a comment can be added to the snapshot. The file created has a default extension of `.snap`. Snapshot files can be used as initial condition files (see Section 12.6).

Apply Snapshot
This menu item will have a sub-menu showing all available initial condition and snapshot files, i.e. all files referenced within the current simulation definition. Select one of the initial conditions to override current simulation values with the values in that file.

Apply Initial Condition
Apply the selected initial condition file to the currently active simulation to override the current simulation values with the values from the selected file.

[2]As a last resort, use the `efoKill` command from a UNIX shell or Windows NT command prompt to remove the remaining executables, see Section 14.7.2. The `efoList` command can be used to list the simulator runs currently executing on the host machine, see Section 14.7.1 or the UNIX manual pages for more information.

Check Health

Check whether the connection to the simulator is working correctly. A message appears in the log pane describing the health status of the simulator.

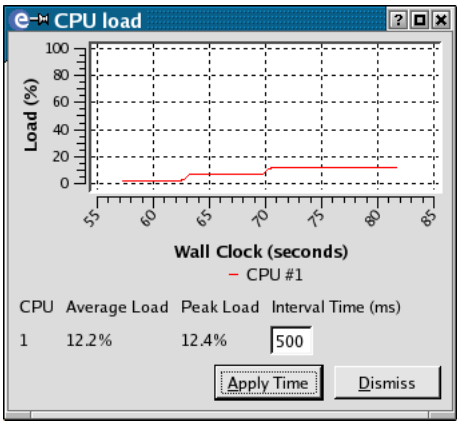
12.7.5 Tools menu

Preferences

Show a preferences dialog for editing global settings. It allows you to specify the maximum number of Simulation Definition files that are stored in the most recently used files list in the *File* menu. You can also select whether all changes are always automatically written to disk when the stimulator is started.

CPU *Load*

This option enables or disables a CPU load monitor as shown in Figure 12.9.



CPU	Average Load	Peak Load	Interval Time (ms)
1	12.2%	12.4%	500

Figure 12.9: The CPU load window

The average and peak load percentage readings are shown for each CPU. The loads are measured over the time interval specified in the line edit in the last column. The time interval can be set in a range from 1 to 9999 ms. If you edit values in the last column you should press the **Apply Time** button to actually use the changed value. The graphical plot shows the peak values measured in the specified interval.

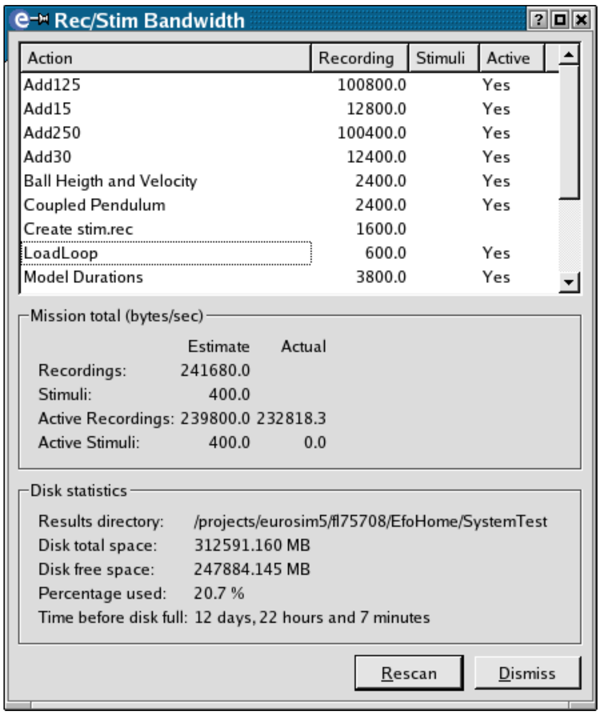
This CPU load monitor is only available if a connection to a simulator is active and the simulator is running in real time.

Rec/Stim Bandwidth

This menu item will show in a window (see Figure 12.10) the runtime bandwidth (in bytes/second) for the recorders and stimuli defined in all scenarios in the Simulation Definition. There are two estimates: one for all actions and one for all *active* actions. These estimates do not take into account start and stop times of these actions, or any other conditions (such as a test like `if varx >100 record ...`). The *actual* bandwidth values are only available during a simulation.

The *Time before disk full* item is an estimate based on the bandwidth of the active recorders and does not take other file actions into account. It also assumes that all recorder files are written to the results directory as displayed in this window.

Press the Rescan button to perform a new calculation based on the most actual bandwidth and free disk space values.



Action	Recording	Stimuli	Active
Add125	100800.0		Yes
Add15	12800.0		Yes
Add250	100400.0		Yes
Add30	12400.0		Yes
Ball Heigth and Velocity	2400.0		Yes
Coupled Pendulum	2400.0		Yes
Create stim.rec	1600.0		
LoadLoop	600.0		Yes
Model Durations	3800.0		Yes

Mission total (bytes/sec)

	Estimate	Actual
Recordings:	241680.0	
Stimuli:	400.0	
Active Recordings:	239800.0	232818.3
Active Stimuli:	400.0	0.0

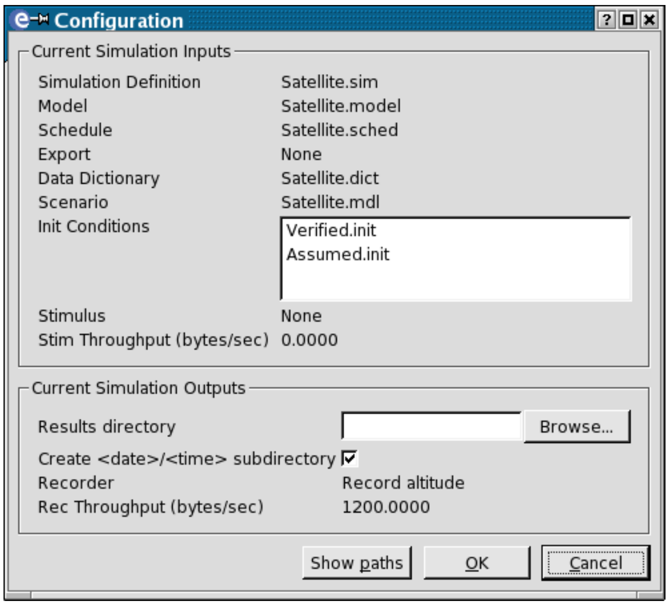
Disk statistics

Results directory:	/projects/eurosim5/fl75708/EfoHome/SystemTest
Disk total space:	312591.160 MB
Disk free space:	247884.145 MB
Percentage used:	20.7 %
Time before disk full:	12 days, 22 hours and 7 minutes

Figure 12.10: The Rec/Stim bandwidth window

Configuration

This menu item will display a window in which various information on the current simulation is given (see Figure 12.11). In the top half of the window the names of the files currently in use as model, schedule, export, data dictionary, initial condition and scenario are displayed, as well as any stimuli data files referenced so far. Finally, the actual stimuli throughput (in bytes/sec) is given. In the bottom half of the window any recording data files in use and the recording throughput are given. Also (prior to requesting Init), the user can change here the directory in which all results files should be stored, as well as whether additional date and time subdirectories should be created where the results files are placed. The **Show Paths** button can be used to view the full path of each of the filenames. The **Rescan** button can be used to get the latest information on the throughput rates.



Current Simulation Inputs

Simulation Definition	Satellite.sim
Model	Satellite.model
Schedule	Satellite.sched
Export	None
Data Dictionary	Satellite.dict
Scenario	Satellite.mdl
Init Conditions	Verified.init Assumed.init
Stimulus	None
Stim Throughput (bytes/sec)	0.0000

Current Simulation Outputs

Results directory	
Create <date>/<time> subdirectory	☑
Recorder	Record altitude
Rec Throughput (bytes/sec)	1200.0000

Figure 12.11: Sample Configuration

12.8 Input Files tab page

This tab page lists all files used in the Simulation Definition. These files can be removed through *Edit:Delete*, new files can be added through the *Insert* menu and the contents can be edited (where applicable) through the *Edit:Properties* menu.
The tab page consists of a tree structure that organizes the files by type:

Top Level
: Shows the used simulator definition (`.sim`), model (`.model`), schedule (`.sched`) and export (`.exports`) files.

Scenarios
: Shows all scenario (`.mdl`) files.

MMI*s*
: Shows all Man-Machine Interface (`.mmi`) files.

Initial Conditions
: Shows all initial condition (`.init`) files.

User Program Definitions
: Shows all User Program Definition (`.usr`) files.

You can reorder the scenario or MMI tab pages. To do that you drag and drop a scenario or MMI file to before or after another scenario or MMI file.
To reorder the Initial Condition files (and thus the order in which these files are applied) you can also use drag and drop to move then around.

12.8.1 Menu items

The following *File* menu items are available in the Input Files tab page:

Select Model
: Select another model file for this Simulation Definition.

Select Schedule
: Select another schedule file for this Simulation Definition.

Select Export
: Select an exports file for this Simulation Definition.

Save File As
: Save the selected file to another location.

The following *Edit* menu items are available in the Input Files tab page:

Properties
: Allows you to edit the properties of the selected file. For scenario and MMI files the corresponding tab page will be raised to the front. For Initial Condition and User Program Definition files a dialog will appear.

Delete
: Remove this file from the Simulation Definition. Note that the actual file is not deleted, the entry is only removed from the Simulation Definition.

Activate
: Only valid for Scenario, MMI and Initial Condition files. Mark this file *Active*, i.e. this file will be used when the simulator starts.

Deactivate
: Only valid for Scenario, MMI and Initial Condition files. Mark this file *Inactive*, i.e. this file will not be used when the simulator starts. Inactive scenario, MMI and initial condition files are ignored by the simulator.

Launch Only valid for User Program files. This will launch the program definition.

If the launch User Program produces output and/or error messages then a window will pop up that shows those messages.

The following *Control* menu item is available in the Input Files tab page:

Apply Initial Condition
The currently selected initial condition file will be applied to the running simulation.

Double clicking on the file name has the same effect as selecting *Properties* from the *Edit* menu. There are a few exceptions: double clicking on a User Program Definition file when a connection to the Simulator is active will *Launch* the program.

12.8.2 Context menus

Two context menus are available in the Input Files tab page depending on where you click the right mouse button. If you click on a file item in the tree then a context menu with the following items appears (see Section 12.8.1 for a description of the menu items):

- Properties
- Delete
- Activate
- Deactivate
- Launch
- Apply Initial Condition
- Select Model
- Select Schedule
- Select Export

The other context menu appears when you click outside the tree area to the right of the last column or below the last row (see Section 12.7.2 for a description of the menu items):

- New Scenario
- Add Scenario
- New MMI
- Add MMI
- New Initial Condition
- Add Initial Condition
- New User Program Definition
- Add User Program Definition

12.9 Schedule tab page

The schedule used by the simulation definition can be debugged in the Schedule tab page (see Section 12.9.1).

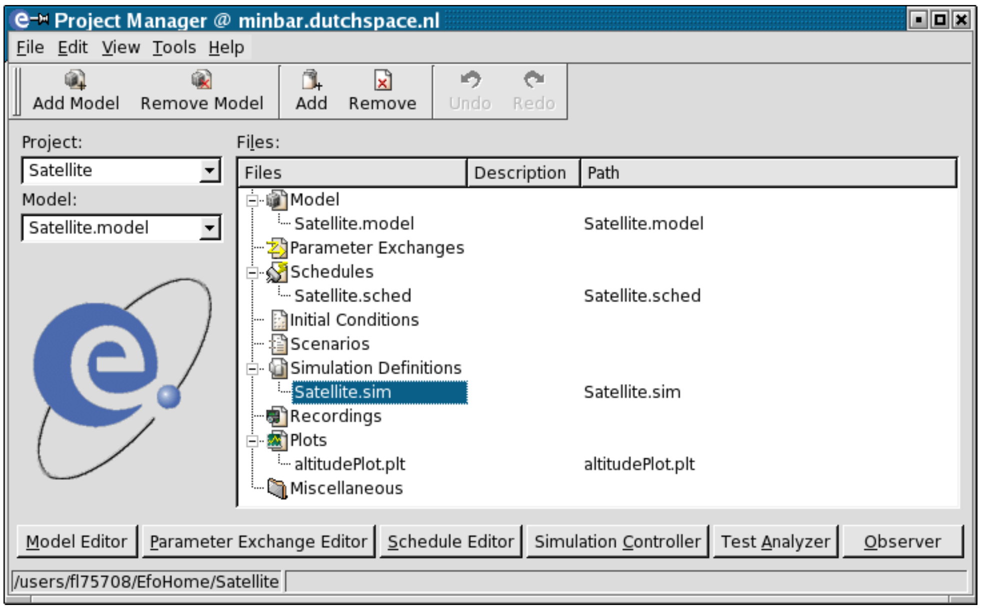


Figure 12.12: The Schedule Tab Page

12.9.1 Debugging Concepts

Debugging a simulation run (or software in general) is a means to investigate why the simulation run is not running as intended. In EuroSim this is done by allowing the user to run the simulation entrypoint for entrypoint. Thus, instead of going through the whole of the simulation, the Debug Control window allows the user to stop at any entrypoint he wishes, or even, to stop at every entrypoint before executing it. This process is called *single stepping* through the simulation code. However, as it can be rather tedious to single step through all entrypoints, *breakpoints* are available. A breakpoint is a kind of stop sign next to an entrypoint. Whenever the simulator encounters such a stop sign, it will hand over control back to the user.

Also, in order to assist the user in debugging the simulation run, entrypoints can be traced and complete tasks can be disabled or enabled at will (note that if a task is disabled, all tasks connected to it 'downstream' in the schedule will also not be called).

Single stepping, breakpoints and disabling of tasks are all easily controlled through the schedule tab page. The schedule tab shows the schedule as defined by the Schedule Editor. You can set breakpoints, traces and enable/disable tasks using the *Debug* menu or by right-clicking on a task to show the context menu.

If you are in debugging mode, then the simulation state is 'executing', even if you are paused at a breakpoint. In such a case, the main window will say 'executing' whilst the simulation time is stopped. In order to return to normal executing, you need to clear all breakpoint tags and continue using the **Continue** button.

If you set a breakpoint of a task in Initializing state, then that breakpoint will not work because the list of breakpoints is passed on to the simulator *after* the Initializing tasks have been called. This is a known limitation.

12.9.2 Debug Control objects

12.9.2.1 ○ Enabled task

These are the tasks as defined in the schedule of the simulation. An enabled task will be executed by the simulator.

12.9.2.2 Disabled task

A disabled task will not be executed by the simulator. Note that any task connected to a disabled task will also not be executed.

12.9.2.3 Current task

The current task (shown in green) is the task currently being executed by the scheduler. If the simulation is run on more than one processor, more than one current task can be present in the schedule view.

12.9.2.4 Breakpoint

This is used to indicate the entrypoint(s) which have a breakpoint attached.

12.9.2.5 Trace

This is used to indicate the entrypoint activation will be traced. A traced entrypoint writes time-tagged messages in the Simulation Controller log window. If an entrypoint has both a trace and a breakpoint, only the breakpoint is shown.

12.9.2.6 Color coding

The tasks are color coded:

blue indicates the selected task.

green indicates the currently executing task/breakpoint.

12.9.3 Menu items

The following *Debug* menu item is available in the scenario tab page:

Item Debug Settings…
Open the Debug Settings window to set and clear breakpoints and traces for the selected task.

Clear All Breakpoints
Clear all breakpoints in the schedule.

Clear All Traces
Clear all traces in the schedule.

Toggle Task Activity
Enable or disable the task.

Continue
Let the simulator run until a breakpoint is encountered. Note that the **Go** button on the main Simulation Controller window cannot be used for this purpose. If **Continue** is requested after all breakpoints have been cleared, then this puts the simulation run back into a normal, non-debugging mode. You can use the function key F8 to quickly access this menu item.

Step Advance the simulation to the next entrypoint to be executed. This button should not be confused with the **Step** button on the Simulation Controller window itself. You can use the function key F10 to quickly access this menu item.

12.10 External debugging facilities[3]

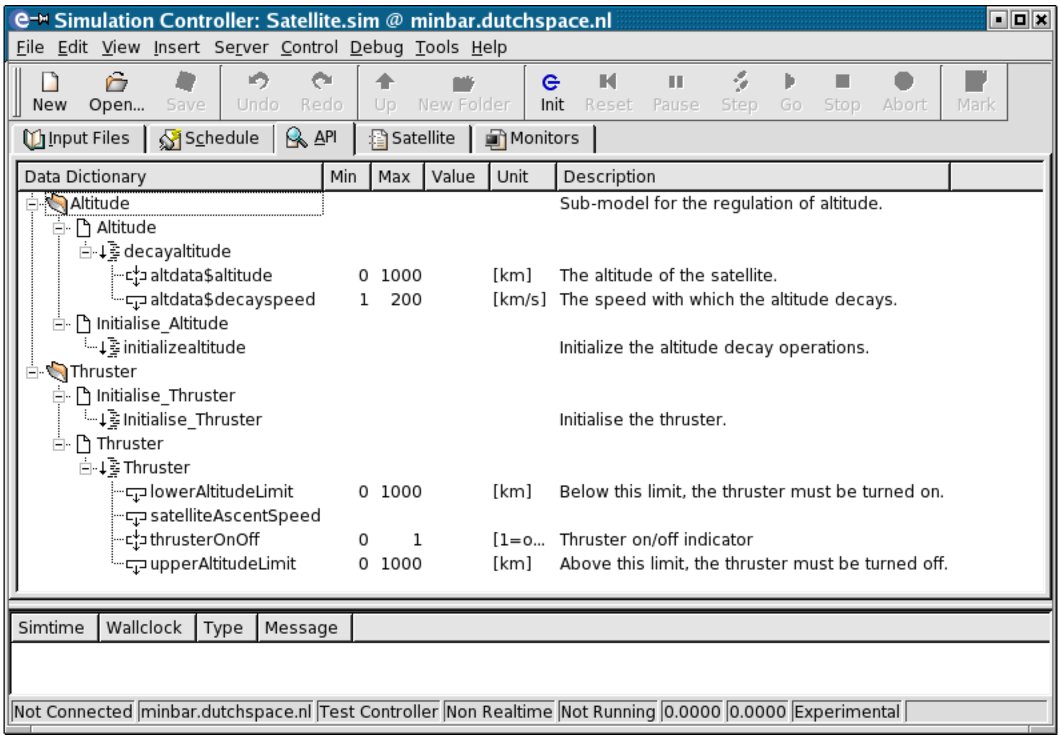
There are two options for debugging model code within EuroSim. The first option is to use the debug control window in the Simulation Controller (see Section 12.9.1). This is useful for tracing which tasks and entrypoints get executed etc. It also offers an integrated interface with EuroSim itself.
However, sometimes it might be necessary to have more control over the executing simulation. In these cases, it is possible to use an external (symbolic) debugger. The only precautions to be taken are to restrict the simulator to one processor (which can be set in the Schedule Editor, see Section 11.3.5), and to use the `-g` flag when building the simulator.
Now the simulation can be started as normal. The debugger can now be connected to the simulator using the command

`# dbx -p`*nnnn*

where *nnnn* is the process number (which can be obtained with the `ps` command. After the connection has succeeded, the Simulation Controller will stop. It will resume when you enter the `cont` command in the debugger. The debugger can be used as on any other application.
The `cvd` debugger can also be used (instead of `dbx`) if the `SIGUSR` and `SIGPOLL` traps are disabled. Both debuggers have to run with 'root' privileges.

12.11 API tab page

The API tab page is a Dictionary Browser (see Section 12.5) with some extra functionality. When no simulation is running it just shows the dictionary with a few extra columns to show the minimum and maximum values, the unit of the value, and the description of the variable.
The column *Value* is empty until a simulation is started. As long as a connection to the simulator is active this column will show the current value of that variable just like a monitor in an MMI tab page. By clicking on the value or by selecting the line and pressing F2 you can edit it and set the variable to a new value. Parameter variables cannot be set as they are read-only. Basically the API tab page is a quick monitor facility.


Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API Satellite Monitors

Data Dictionary	Min	Max	Value	Unit	Description
Altitude					Sub-model for the regulation of altitude.
Altitude					
decayaltitude					
altdata$altitude	0	1000		[km]	The altitude of the satellite.
altdata$decayspeed	1	200		[km/s]	The speed with which the altitude decays.
Initialise_Altitude					
initializealtitude					Initialize the altitude decay operations.
Thruster					
Initialise_Thruster					
Initialise_Thruster					Initialise the thruster.
Thruster					
Thruster					
lowerAltitudeLimit	0	1000		[km]	Below this limit, the thruster must be turned on.
satelliteAscentSpeed					
thrusterOnOff	0	1		[1=o...	Thruster on/off indicator
upperAltitudeLimit	0	1000		[km]	Above this limit, the thruster must be turned off.

Simtime	Wallclock	Type	Message

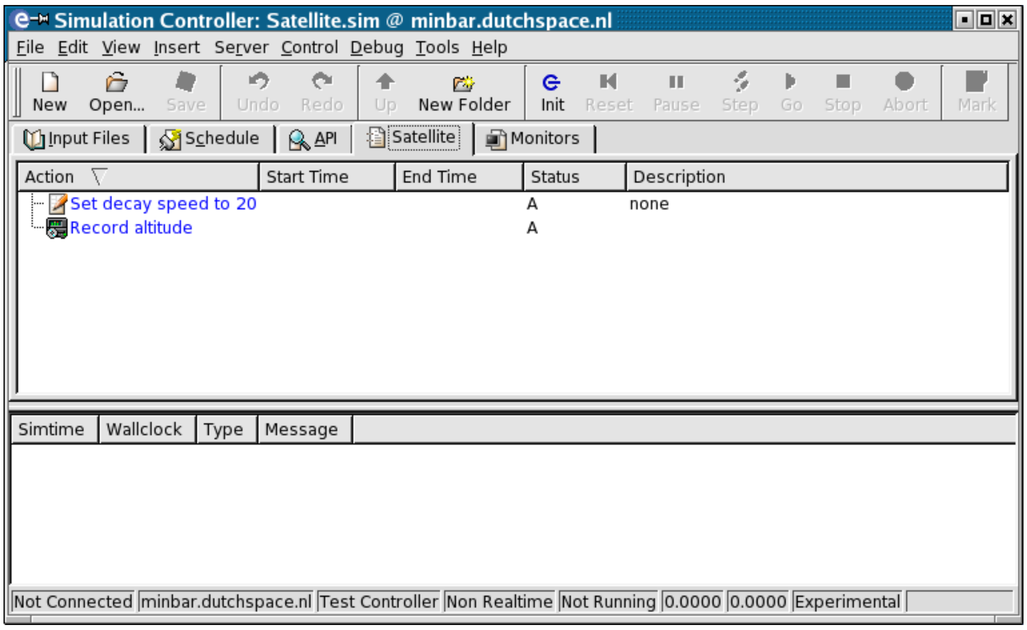
Not Connected | minbar.dutchspace.nl | Test Controller | Non Realtime | Not Running | 0.0000 | 0.0000 | Experimental

Figure 12.13: The API tab page

[3]Not supported on the Windows NT platform.

12.12 Scenario tab page

For each scenario file a separate Scenario tab page is created. When the scenario file is opened or created you are asked to provide the caption that appears as the name of the tab page.
The scenario can be presented either as a tree view (see Figure 12.14) or as an icon view (see Figure 12.15). In both cases the actions in the scenario can be organized in folders.

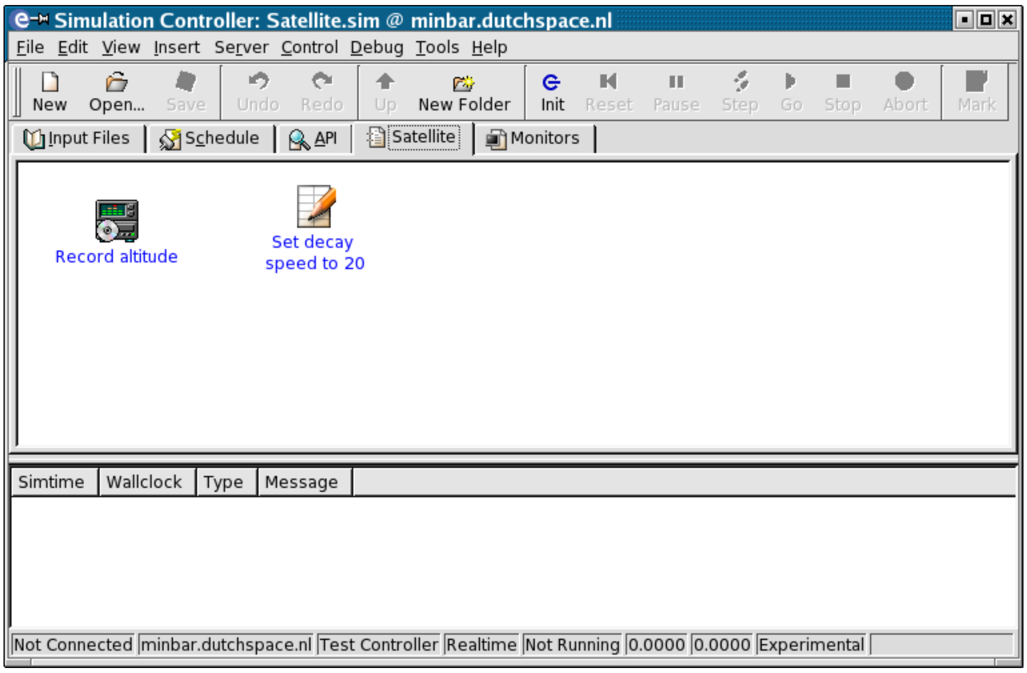

Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API Satellite Monitors

Action	Start Time	End Time	Status	Description
Set decay speed to 20			A	none
Record altitude			A	

Simtime	Wallclock	Type	Message

Not Connected | minbar.dutchspace.nl | Test Controller | Non Realtime | Not Running | 0.0000 | 0.0000 | Experimental

Figure 12.14: The Scenario tab page (tree view)


Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API Satellite Monitors
Record altitude
Set decay speed to 20

Simtime	Wallclock	Type	Message

Not Connected | minbar.dutchspace.nl | Test Controller | Realtime | Not Running | 0.0000 | 0.0000 | Experimental

Figure 12.15: The Scenario tab page (icon view)

Actions in the scenario tab page can be either active or inactive (indicating whether it will be automatically checked against its run condition during a simulation run). For active actions the action name is shown in blue instead of black and (for the tree view only) the last column *Status* is marked with an ‘*A*’.
By toggling the *Active* checkbox in the Action Editor dialog you can change the initial Active state.
During a simulation you can activate an inactive action or deactivate an active action. This does *not*

modify the *Active* property of the action. When the simulation ends the Active status returns to its original setting.
When an action is actually executing, the *Status* column is marked with an '*X*' (for the tree view only) and the action name is shown in green instead of blue (active action) or black (inactive action).
Icons are used to represent actions (stimuli, recorders, monitors, scripts) or folders. The following icons are used in the scenario tab page:

Recorder
: this icon is used for recorder actions (defined using the Recorder Editor)

Stimulus
: this icon is used for stimulus actions (defined using the Stimulus Editor)

Monitor
: this icon is used for monitor actions (can only appear in old pre-Mk.3 scenario files)

Script
: this icon is used for script (free format MDL) actions

Folder
: this icon is used for folders that can contain other actions or folders.

Double clicking on these actions when a simulation is running will have the following effect depending on the type of action:

Recorder
: activate or deactivate this recorder

Stimulus
: activate or deactivate this stimulus

Monitor
: start this monitor (it will show up on the *Script Monitors* tab page)

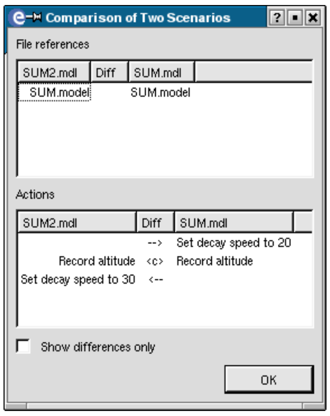
Script trigger this action

You can drag and drop actions and folders from one place to another. In order to rename a folder or action you can click on the item with the left mouse button to select it, then click again to edit the name. You can also press *F2* to edit the name of the selected item.

12.12.1 Menu items

The following *File* menu item is available in the scenario tab page:

Diff with
: This menu option will pop-up a file-selection box, in which another scenario file can be selected. The selected scenario file will be compared with the current file, and any differences will be reported. The following symbols are used to identify any differences; these will appear between column listings of components in scenario A (first column) and scenario B (second column): -> means that an item is present in B but not in A <- means that an item is present in A but not in B <-> means that there is a difference in versions between a file in both scenarios <b> means that there is a difference in the body of two actions with the same name <c> means that there is a difference in the condition of two actions with the same name. See Figure 12.16 for an example.



File references

SUM2.mdl	Diff	SUM.mdl
SUM.model		SUM.model

Actions

SUM2.mdl	Diff	SUM.mdl
	-->	Set decay speed to 20
Record altitude	<c>	Record altitude
Set decay speed to 30	<--	

Figure 12.16: Example difference list

The following *Edit* menu items are available in the scenario tab page:

Undo/Redo
: Action changes and changes to the hierarchy structure of a scenario (i.e. actions moved to another folder, folders dragged to another position, folders deleted or added) can be undone and redone.

Cut/Copy/Paste
: Actions and folders support the usual cut, copy and paste operations. An action/folder that is copied or cut from one scenario tab page can be pasted onto the tab page of another scenario.

Activate/Deactivate
: Activate or deactivate the selected action. Only available if a simulation is running.

Properties
: Start the editor for the selected action.

Delete
: Delete the selected action or folder. The action or folder is not placed in the clipboard and thus cannot be pasted.

Edit Scenario Caption
: Change the caption of the scenario tab page.

Delete Scenario Tab Page
: Delete the scenario tab page. You will be asked to confirm this operation.

The following *Edit* menu items are available in the scenario tab page:

Show Icon View
: Toggle between the tree view and the icon view of the scenario.

Rearrange Icons
: Icon view specific: rearrange the icons of the scenario.

Up
: Icon view specific: by double clicking on a folder you move down in the action hierarchy. This menu item moves the icon view to one level up the action hierarchy.

The following *Insert* menu items are available in the scenario tab page:

New Recorder
: Create a new recorder action. See Section 12.13.2 for more information.

New Stimulus
: Create a new stimulus action. See Section 12.13.3 for more information.

New Script
: Create a new script action. See Section 12.13.1 for more information.

New Folder
Create a new folder called *New Folder* followed by a unique number. You can immediately edit the generated folder name and change it to something more appropriate.

The following *Control* menu item is available in the scenario tab page:

Execute Action
Execute the selected action. Only available when the connection to the simulator is active.

The following *Tools* menu items are available in the scenario tab page:

Commandline Script
Quickly enter an action script and execute it. Only available if there is a connection to a simulator.

Convert Old Monitors
Convert all monitor actions in this scenario to a new MMI tab page. You are asked for the filename of the new `.mmi` file, the caption for the new tab page and if you want to delete the old monitors after conversion.

12.12.2 Context menus

Two context menus are available in the Scenario tab page depending on where you click the right mouse button. If you click on an action item in the tree then a context menu with the following items appears (see Section 12.12.1 for a description of the menu items):

- Properties
- Activate
- Deactivate
- Execute Action
- Delete
- Cut
- Copy
- Paste
- Undo
- Redo

The other context menu appears when you click outside the tree area to the right of the last column or below the last row (see Section 12.12.1 for a description of the menu items):

- New Recorder
- New Stimulus
- New Script
- New Folder
- Up
- Paste
- Undo

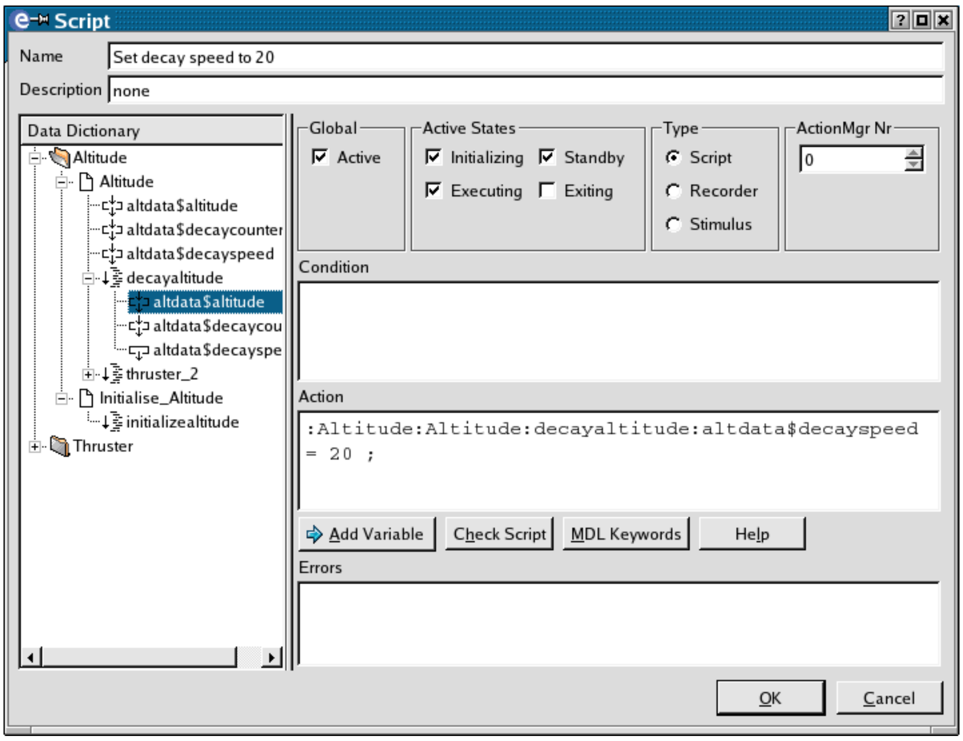
- Redo
- Rearrange Icons
- Edit Scenario Caption
- Delete Scenario Tab Page

12.13 Action Editor reference

The Action Editor allows for the creation and modification of action objects, as they are used in the Simulation Controller. For each of the three possible action types, a variation of the Action Editor is used. A number of elements are shared amongst all editor variations, and these are described in the section on script actions (Section 12.13.1).
All actions are ultimately defined in MDL and handled at run-time in the same way. The provision of the Action Editors is to allow the most common types of actions to be created with the minimum knowledge of MDL syntax.

12.13.1 Script Action Editor

The script Action Editor is shown in Figure 12.17.



Name: Set decay speed to 20

Description: none

Global: [✓ Active]

Active States: [✓ Initializing] [✓ Standby] [✓ Executing] Exiting

Type: [✓ Script] Recorder Stimulus

ActionMgr Nr: 0

Action:

```
:Altitude:Altitude:decayaltitude:altdata$decayspeed
= 20 ;
```

Figure 12.17: The Script Action Editor

The window consists of several parts, each part corresponding to an element of an action, as described in Section 12.2.2. In the first three parts, the following attributes can be entered:

Action name
: This is the name of the action as it appears in the tree or icon view. It should be a unique name within the current scenario.

Description
: A description of the action.

Global & Active States
: These options are used to indicate whether the action should either be active or inactive when the scenario is started; as well as in which of the four simulation states the action should be active.

ActionMgr Nr
: This attribute allows you to specify on which action manager this action will be executed.

The next part of the window is a text entry area where the execution condition of the current action can be specified. The execution condition is specified using the Mission Definition Language (see Appendix E).
The final part of the window is another text entry area in which the actual action script can be entered. The **Check script** button can be used to check whether or not the entered MDL scripts are syntactically correct.
The **MDL Keywords** button will pop up a small window with a list of all available MDL commands. With the **Add to Clipboard** button (or by double clicking on a command) you can copy the command to the clipboard and paste it in the Condition or Action text entry areas.
The **Events** button will show a window with all input connectors from the schedule. With the **Add to Clipboard** button (or by double clicking on an events) you can copy the events to the clipboard and paste it in the Condition or Action text entry areas. If no user defined input connectors are found, then this button will not appear.
Any errors that are detected in the condition or action text will appear in the *Errors* area at the bottom of the window.
The left hand side of the window contains a Dictionary Browser (see Section 12.5) that you can use to drag and drop variables from the dictionary to the condition or action text areas. You can select more than one variable and they will be inserted into the text as a list of variables, one per line.
Besides drag and drop you can also double click on a variable to add it at the current cursor position, or use the **Add Variable** button to add all selected variable at the current cursor position.

12.13.2 Recorder Action Editor

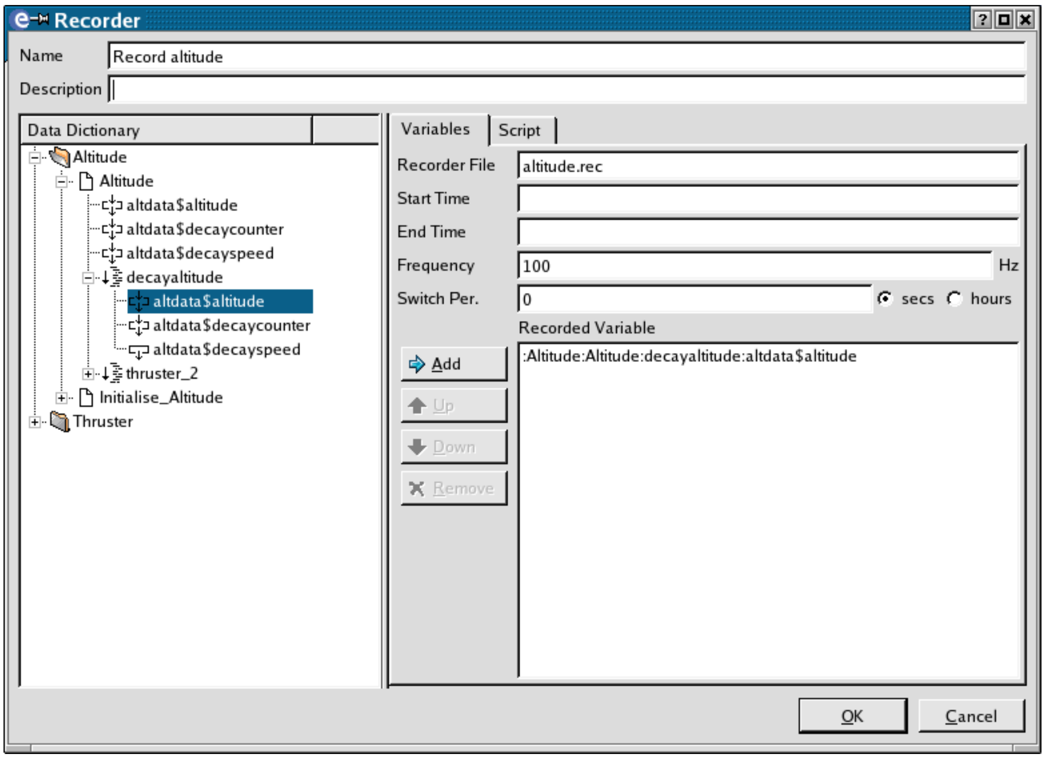
The recorder Action Editor consists of two tab pages. The editor with the first tab page (*Variables*) on top is shown in Figure 12.18. The second tab page (*Script*) is the same as the script Action Editor window (Figure 12.17) except for an extra checkbox *Manual*. When checked the Condition and Action text areas can be edited, and the entry fields in the *Variables* tab page cannot be edited. When unchecked the situation is the other way around.
It should not be necessary to check the *Manual* checkbox when building simple recorders. For more complex recorders you could start with the *Variables* tab page, fill in all the fields, switch to the *Script* tab page, check the *Manual* checkbox and then customize the condition and action.
In the *Variables* tab page, the following information can be entered to define a recording action.

Action name and Description
: As for the script action attributes.

Recorder File
: The name of the file in which the recorded variable values should be stored. The default filename is *actionname*`.rec`.

Frequency, Start Time and End Time
: The three attributes specify when the recording should start and stop, and with what sample rate the variable values should be written to the file. Note: if UTC is selected times should entered as YYYY-mm-dd HH:MM:SS[.sss], e.g. 2001-12-31 16:01:02.400.

Switch Per.
: A switch time can be specified whether the recorder should switch periodically. This value is given in units of seconds or hours. After each elapsed switch time the recorder *actionname*`.rec` is closed and *actionname-nnn*`.rec` is opened (with switch counter *nnn*).



Name: Record altitude
Description:
Recorder File: altitude.rec
Start Time:
End Time:
Frequency: 100 Hz
Switch Per.: 0 [✓ secs] hours
Recorded Variable: :Altitude:Altitude:decayaltitude:altdata$altitude

Figure 12.18: The Recorder Action Editor

Below these attributes the *Recorded Variable* listbox is shown. If any variables were added from the Dictionary Browser (see Section 12.5), they are shown here. Variables can be added using drag and drop, by double clicking on a variable in the Dictionary Browser, or by selecting variables in the Dictionary Browser and pressing the **Add** button to add them. To remove a variable from the list, select it, and press the **Remove** button. You can change the order of the variables by selecting variables in the listbox and using the **Up** and **Down** buttons.
The values of the variables in the list are recorded into the specified file at the specified frequency. EuroSim automatically generates an MDL-script for this purpose, which can be viewed in the *Script* tab page. If you want to use a non-numerical start or end time you can change the values manually in that tab. For example, you can use a simulator variable as the end time.

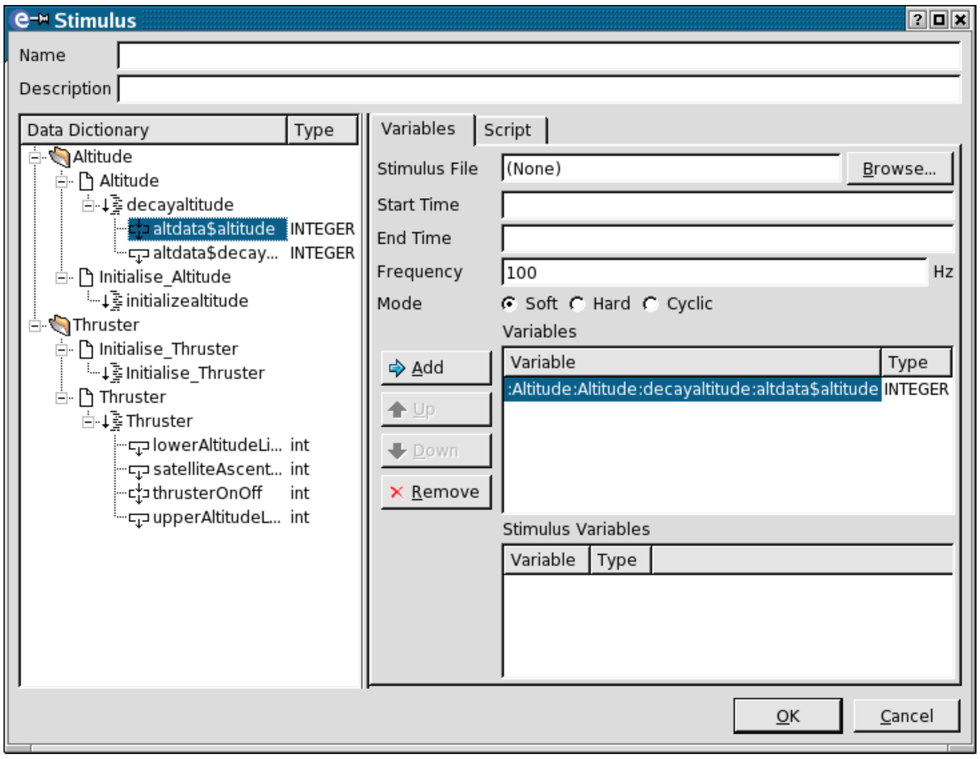
12.13.3 Stimulus Action Editor

When the stimulus editor is started you will be asked to select a stimulus file. You can select both a `.stim` file or a `.rec` recorder file.
The stimulus Action Editor consists of two tab pages (see Figure 12.17 and Figure 12.18). The Script Action Editor tab page (see Figure 12.17) is identical for both cases. The first stimulus Action Editor tab page (see Figure 12.18) has the following fields:

Stimulus File
This should be the name of the input file containing the stimulus data.[4] You can use the **Browse** button to select an input file.

Frequency, Start Time and End Time
The three attributes specify when the stimulus should start and stop, and with what sample rate the variable values should be read from the file. Note: if UTC is selected times should entered as YYYY-mm-dd HH:MM:SS[.sss], e.g. 2001-12-31 16:01:02.400.

[4]Note that this action editor can only be used to make stimuli actions which read in data from an external source. To update a variable using a function (e.g. to feed a sinusoidal value), this needs to be defined using a script Action Editor with e.g. varZ = sin(varX).


Stimulus

Name:

Description:

Data Dictionary	Type
Altitude	
Altitude	
decayaltitude	
altdata$altitude	INTEGER
altdata$decay...	INTEGER
Initialise_Altitude	
initializealtitude	
Thruster	
Initialise_Thruster	
Initialise_Thruster	
Thruster	
Thruster	
lowerAltitudeLi...	int
satelliteAscent...	int
thrusterOnOff	int
upperAltitudeL...	int

Variables | Script

Stimulus File: (None) Browse...

Start Time:

End Time:

Frequency: 100 Hz

Mode: [✓ Soft] Hard Cyclic

Variables

Variable	Type
:Altitude:Altitude:decayaltitude:altdata$altitude	INTEGER

Add Up Down Remove

Stimulus Variables

Variable	Type

OK Cancel

Figure 12.19: The Stimulus Action Editor

Variables
If any variables were added from the Dictionary Browser (see Section 12.5), they are shown here. Variables can be added using drag and drop, by double clicking on a variable in the Dictionary Browser, or by selecting variables in the Dictionary Browser and pressing the **Add** button to add them. To remove a variable from the list, select it, and press the **Remove** button. You can change the order of the variables by selecting variables in the listbox and using the **Up** and **Down** buttons.

Stimulus Variables
The variables you add to the *Variables* list must match with the variables from this list. This list is extracted from the selected stimulus file. The variable types are shown in both lists and in the Dictionary Browser. This makes it easier to find a match. If the *Variables* list is empty when a stimulus file was selected, then the program tries to prefill the *Variables* list with correct matches.

Mode This can either be set to *soft*, *hard* or *cyclic*. With the first option, the data in the stimulus file is read in sequential order at the specified frequency, and the timestamps attached to the data are ignored. With the second option, only those data from the file are used whose timestamp match the current simulation time (or has the nearest elapsed time) when the data is requested. Data between these points are ignored. With the third option the data in the stimulus file is read in sequential order and after the last datapoint read, the stimulus file is reread from the beginning. These stimuli data is applied in 'soft' manner.

Consider the following input data file:
Data file:

```
simtime   data
0.9       10
1.9       15
2.9       17
3.9       19
```

```
4.9      20
5.9      18
6.9      15
7.9      15
8.9      14
9.9      12
```

If the stimulus action is to update variable ‘Z’ at a frequency of 0.5 Hz, and the stimulation mode was set to *soft*, then ‘Z’ would be updated as follows, i.e. every 2 seconds the next value is used from the file:
Simulation:

```
simtime  Z
0        10
2        15
4        17
6        19
8        20
10       18
12       15
14       15
16       14
18       12
20       no more data
```

If the stimulus actions is to update variable ‘Z’ at a frequency of 0.5 Hz, and the stimulation mode was set to *hard*, then ‘Z’ would be updated as follows, i.e. every 2 seconds the most ‘up-to-date’ value is used from the file:
Simulation:

```
simtime  Z
0        0
2        15
4        19
6        18
8        15
10       12
12       no more data
```

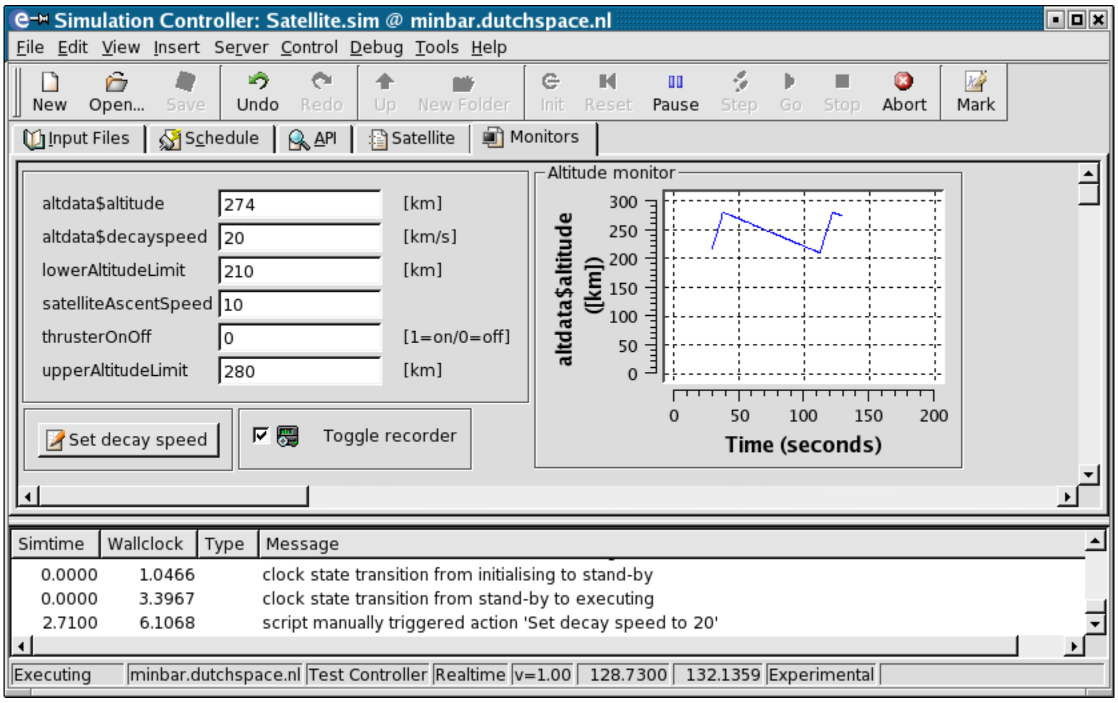
If the stimulus action is to update variable ‘Z’ at a frequency of 0.5 Hz, and the stimulation mode was set to *cyclic*, then ‘Z’ would be updated as follows, i.e. every 2 seconds the next value is used from the file, and when there is no more data, the data from the file is used again:
Simulation:

```
simtime  Z
0        10
2        15
4        17
6        19
8        20
10       18
12       15
14       15
16       14
18       12
20       10 (start from the beginning)
22       15
```

etc.

12.14 MMI tab page

For each `.mmi` file a separate MMI (Man-Machine Interface) tab page is created. When the `.mmi` file is opened or created you will be asked to provide the caption that appears as the name of the tab page.
The MMI tab page is a large pane on which you can place monitors to monitor variables in the simulation. There are two basic types of monitors: alpha numerical, i.e. each variable is presented as a caption followed by the value, and graphical, where each variable is tracked over time (or possibly against another variable) and plotted on a canvas. See Figure 12.20 for an example. Besides monitoring variables you can also add *Action Buttons* to execute MDL scripts or to enable/disable recorders or stimuli.


Simulation Controller: Satellite.sim @ minbar.dutchspace.nl
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API Satellite Monitors
Altitude monitor
altdata$altitude ([km])
300 250 200 150 100 50 0
0 50 100 150 200
Time (seconds)
Set decay speed
Toggle recorder

altdata$altitude	274	[km]
altdata$decayspeed	20	[km/s]
lowerAltitudeLimit	210	[km]
satelliteAscentSpeed	10	
thrusterOnOff	0	[1=on/0=off]
upperAltitudeLimit	280	[km]

Simtime	Wallclock	Type	Message
0.0000	1.0466		clock state transition from initialising to stand-by
0.0000	3.3967		clock state transition from stand-by to executing
2.7100	6.1068		script manually triggered action 'Set decay speed to 20'

```
Executing | minbar.dutchspace.nl | Test Controller | Realtime | v=1.00 | 128.7300 | 132.1359 | Experimental
```

Figure 12.20: The MMI tab page

When you select a monitor by clicking on the monitor window with the left mouse button a rectangle with 'grab handles' appears. By clicking on the handles and moving the mouse around (keeping the left mouse button pressed) you can resize the monitor. If you click inside the rectangle and move the mouse around you can move the monitor to another place.
You can insert a new monitor by using the *Insert:New Monitor* menu item or by double clicking in the MMI tab page. Double clicking on a monitor will open the *Properties* window where you can modify the properties of that monitor.
You can insert a new action button by using the *Insert:New Monitor* menu item. Double clicking on an action button will open the *Properties* window where you can modify the properties of that action button.

12.14.1 Menu items

The following *Edit* menu items are available in the MMI tab page:

Undo/Redo
: When a monitor or action button is resized, moved, or properties are changed then those changes can be undone and redone.

Cut/Copy/Paste
: Monitors and action buttons support the usual cut, copy and paste operations. A monitor or action button that is copied or cut from one MMI tab page can be pasted onto the tab page of another MMI.

You can also (as a special case) copy or cut an old monitor action from a scenario tab and paste it onto an MMI tab page. The reverse is not possible since monitor actions are obsolescent.

Properties
: Edit the properties of the selected monitor or action button.

Copy to Desktop
: Copy the monitor or action button as a floating window on the desktop.

Edit MMI Caption
: Change the caption of the MMI tab page.

Delete MMI Tab Page
: Delete the MMI tab page. You will be asked to confirm this operation.

The following *Insert* menu items are available in the MMI tab page:

New Monitor
: Create a new monitor. See Section 12.14.3 for more information.

New Action Button
: Create a new action button. See Section 12.14.4 for more information.

12.14.2 Context menus

Two context menus are available in the MMI tab page depending on where you click the right mouse button. If you click on a monitor or action button then a context menu with the following items appears (see Section 12.14.1 for a description of the menu items):

- Properties
- Copy to Desktop
- Delete
- Cut
- Copy
- Paste
- Undo
- Redo

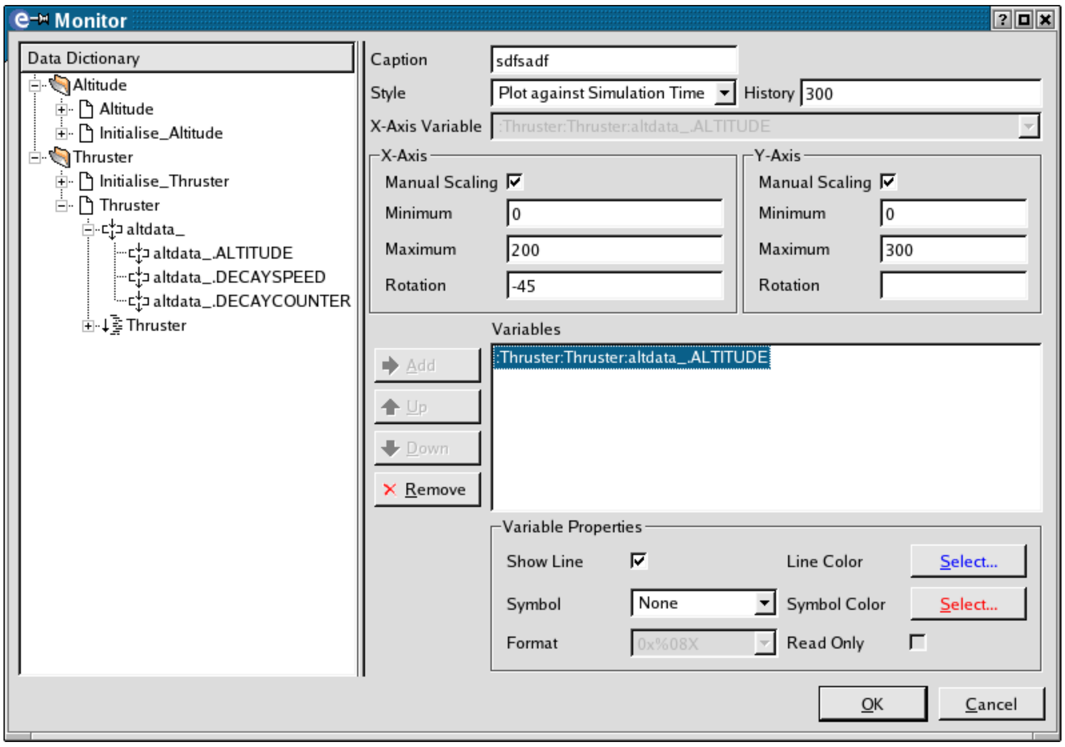
The other context menu appears when you click directly on the tab page background (see Section 12.14.1 for a description of the menu items):

- New Monitor
- New Action Button
- Paste
- Undo
- Redo
- Edit MMI Caption
- Delete MMI Tab Page
- Activate MMI Tab Page
- Dectivate MMI Tab Page

The latter two menu items, *Activate* MMI *Tab Page* and *Dectivate* MMI *Tab Page*, are short-cuts to the *Activate* and *Deactivate* menu items that are available in the *Edit* menu of the Input Files tab page (see Section 12.8.1).

12.14.3 Monitor Editor

The monitor editor is similar to the recorder Action Editor (see Figure 12.18) in terms of overall layout, but there are still many differences.
Nevertheless, as can be seen in Figure 12.21, the basics are the same: on the left hand side is the Dictionary Browser (see Section 12.5 for more information), on the right hand side is a *Variables* list and in between are buttons to add to, remove from and rearrange the variables in the list.
If you try to add an array or structure that contains more than 10 elements you will be asked if this is really what you want. Since structures and arrays are expanded in the *Variables* list to their constituent variables this prevents against the accidental selection of large arrays or structures. A monitor of more than 10 variables is generally not very useful.
There are two property areas in the editor: the properties above the *Variables* list are properties of the monitor as a whole, the properties below the list are properties of the currently selected variable in the *Variables* list.


Monitor

Data Dictionary
- Altitude
  - Altitude
  - Initialise_Altitude
- Thruster
  - Initialise_Thruster
  - Thruster
    - altdata_
      - altdata_.ALTITUDE
      - altdata_.DECAYSPEED
      - altdata_.DECAYCOUNTER
    - Thruster

Caption: sdfsadf
Style: Plot against Simulation Time — History: 300
X-Axis Variable: :Thruster:Thruster:altdata_.ALTITUDE

X-Axis
Manual Scaling: ✓
Minimum: 0
Maximum: 200
Rotation: -45

Y-Axis
Manual Scaling: ✓
Minimum: 0
Maximum: 300
Rotation:

Variables
:Thruster:Thruster:altdata_.ALTITUDE

Variable Properties
Show Line: ✓ — Line Color: Select...
Symbol: None — Symbol Color: Select...
Format: 0x%08X — Read Only: ☐

Figure 12.21: The Monitor Editor

12.14.3.1 Monitor Properties

The following properties are always available:

Caption
Enter the caption of the monitor.

Style Select the style of the monitor. The following styles are available:

Alpha Numeric
Give a textual representation of the value of a variable.

Plot against Simulation Time
Use the value of the variable as the Y-axis value and the simulation time as the X-axis value.

Plot against Wall Clock Time
Use the value of the variable as the Y-axis value and the wall clock time as the X-axis value.

XY-Plot Use the value of the variable as the Y-axis value and the value of a designated other variable as the X-axis value.

Depending on the style some of the other properties in the monitor editor become enabled or disabled. For the Alpha Numeric style the *Read Only* checkbox in the variable properties area is only enabled if the variable is an input variable and the *Format* combobox is only enabled if the variable is not a string. For the plot styles all properties are enabled except for the *Read Only* checkbox and the *Format* combobox. The *X-Axis Variable* combobox is only enabled when the *XY-Plot* style is selected.

The following properties are available when one of the plot styles is selected:

History This value indicates how many samples of each variable should be simultaneously displayed. Once the maximum is reached, the older values will be discarded.

Manual scaling
This checkbox can be checked if the user wishes to specify the minimum and maximum values for the axis.

Minimum
The minimum value for the corresponding axis.

Maximum
The minimum value for the corresponding axis.

Rotation
The rotation of the labels on the corresponding axis.

The following property is available when the *XY-Plot* style is selected:

X-Axis Variable
Select a variable from the *Variables* list that provides the X-Axis variable values.

12.14.3.2 Variable properties

The variable properties are disabled if no variable is selected in the *Variables* list. Otherwise they change the representation of the selected variable.

The following properties are available when the *Alpha Numeric* style is selected:

Format Allows you to enter an optional formatting string using the printf style, see Section 12.14.3.3. The drop down list box gives you a few suggestions for representing integer values as hexadecimals.

Read Only
If checked, then this variable cannot be modified in the monitor.

During a simulation run, an alphanumeric monitor can be used as a mechanism for updating the value of the variable(s) it is displaying. You just need to type a new value into the field and press Return. If the *Format* field specifies a conversion, f.i. to hexadecimal, then you must also enter the value in that format. For traceability, this update event is logged. Read-only variables cannot be edited and are displayed as text instead of an edit field. If the variable is a parameter, then that variable is always read-only.

The following properties are available when a Plot style is selected:

Show Line
If checked, connect the data points in the plot with a line.

Line Color
Press the **Select...** button to select the color for the line.

Symbol Choose a symbol to be used for each data point.

Symbol Color
Press the **Select...** button to select the color for the symbol.

12.14.3.3 Variable formatting and conversion

The *Format* field of the Variable properties allows formatting and/or conversion of the monitored variable. When this field is left blank, then a default formatting will be applied that is appropriate for the type of the variable. The *Format* field supports a sub-set of the format string as specfied for the *printf* function, see the `printf(3)` man page for more details.

The following length modifiers are supported: **h** (short int or unsigned short int), **ll** (long long int or unsigned long long int). Make sure that the length modifier matches the type of the model variable in the simulator. You can retrieve the variable type by pressing the right mouse button on the variable in the Dictionary Browser and selecting the Info menu item in the context menu. Variables of type int, long int, float and double do not need a length modifier in the format string (note that int and long int are the same on 32-bit platforms).

The following conversion specifiers are explicitly *not* supported: **c** (character) and **s** (string). Table 12.1 gives a few examples of formatting and conversion of monitored variables. Note that conversion to/from hexadecimal values can only be done on integers, while formatting of floating point numbers only works on float and double types.

Value in simulator	Format	Result in monitor
255	%X	FF
255	%08X	000000FF
255	0x%08X	0x000000FF
3.141592	%.2f	3.14
3000	%.2E	3.00E+03

Table 12.1: Examples of formatting and conversion.

12.14.4 Action Button Editor

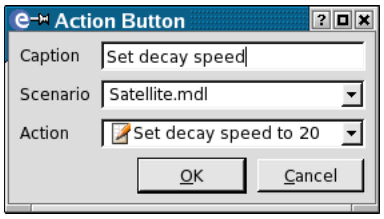
The Action Button Editor (see Figure 12.22) allows you to add a button or checkbox to the MMI pane to execute MDL scripts or enable/disable recorders or stimuli. The editor has the following properties:

Caption
: This is the text that you want on the button/checkbox. If left empty, then the name of the action is used instead.

Scenario
: Choose the scenario containing the action that you want to use.

Action
: Choose the action from the scenario selected above.

A script action will now appear on the MMI tab as a button. Pressing the button when simulator is running will execute the action. Recorders and stimuli appear as a checkbox. When checked the recorder or stimulus is active, when unchecked it is not active. Toggling the checkbox will activate/deactivate the recorder or stimulus. See Figure 12.20 for an example.



Caption: Set decay speed
Scenario: Satellite.mdl
Action: Set decay speed to 20

Figure 12.22: The Action Button Editor

Chapter 13

Test Analyzer reference

The Test Analyzer can be used to create and display plots of the generated test results. It uses PV-WAVE [1] or gnuplot to display and print the plots. For most plots the user interface of the Test Analyzer is sufficient, but it is also possible to send commands to the PV-WAVE or gnuplot back-end directly.
The purpose of this chapter is to provide a detailed reference of the Test Analyzer.
The first part of this chapter describes how to start and use the Test Analyzer (Section 13.1 - Section 13.2).
The second part can be used for reference (Section 13.4 - Section 13.7).

13.1 Starting the Test Analyzer

The Test Analyzer can be started by selecting the Test analyzer button in the EuroSim start-up window.
The Test Analyzer can also be started from the command line by issuing the `TestAnalyzer` command.

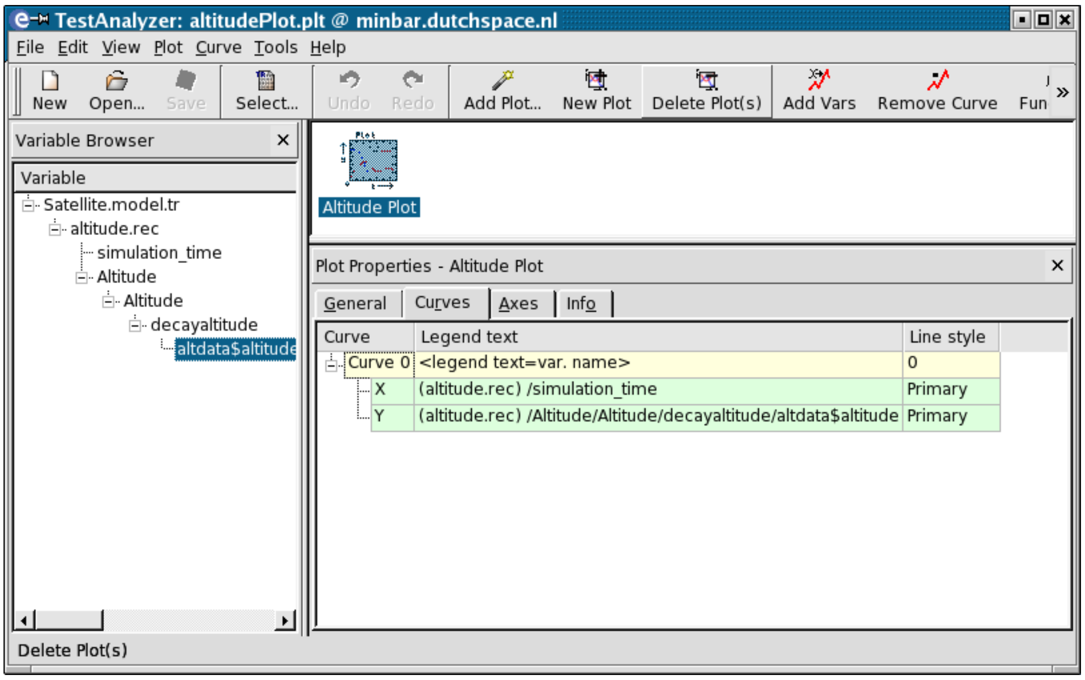
13.2 Using the Test Analyzer

The next sections describe how the Test Analyzer can be used without going into too much detail. For a complete description of a particular part of the user interface please refer to Section 13.4 - Section 13.7.

13.3 Test Analyzer main window

The main window of the Test Analyzer is shown in Figure 13.1. The main window contains the following elements:

[1] Not supported on the Windows NT platform.



Variable
Satellite.model.tr
altitude.rec
simulation_time
Altitude
Altitude
decayaltitude
altdata$altitude

Curve	Legend text	Line style
Curve 0	<legend text=var. name>	0
X	(altitude.rec) /simulation_time	Primary
Y	(altitude.rec) /Altitude/Altitude/decayaltitude/altdata$altitude	Primary

Figure 13.1: The Test Analyzer main window

Menu bar
: For a detailed description of the menu items see Section 13.7.

Toolbar A description of the action the toolbar button performs is displayed if the mouse is left above the button for a short period of time. The toolbar provides a shortcut to many often used menu items like undo, redo, add plot, etc.

Plot view
: The plot view holds the icons representing the plots that are defined.

Variable browser
: The variable browser contains the variables found in the test results that are loaded. You can use these variables to create or edit curves in the plots.

Plot properties
: The plot properties pane contains three tabpages. The first page deals with the general plot properties like plot title and description. The second page is dedicated to the curves of the plot (*curve editor*). The third page is used to change axes related settings like scaling (linear/logarithmic) and axis range.

Statusbar
: The status bar displays the location of the currently loaded test results file on the right. The rest of the statusbar is used to show short (status) messages.

13.3.1 Opening a plot file

The Test Analyzer works with plot files. A plot file contains one or more (often related) plots. Previous versions of the Test Analyzer worked with plot definition files (pdf). This file format is no longer in use. Instructions on how to convert old pdf files can be found in Section 13.3.2.

To open a plot file, select *File:Open...* from the menu or click on the button on the toolbar. The plot view now shows the plots defined in this file. To be able to show the plots, test results need to be loaded as well.

13.3.2 Importing old plot definition files

To import old plot definition files, select *File:Open*. In the dialog that appears, select the "Plot definition files (*.pdf)" from the file filter selection area (see picture below).

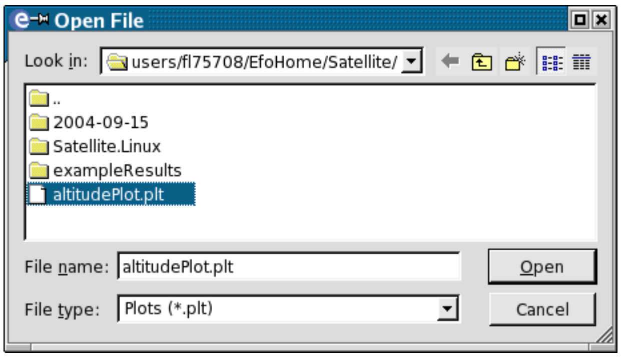


Figure 13.2: Importing plot definition files. Click on the "File type" combobox to switch between file formats.

Next, browse to the plot definition file that needs to be imported and click on the **OK** button. A warning message will appear stating that the pdf file will be converted. Press **OK** to convert the pdf file.
The Test Analyzer now contains the converted data. If you wish you can save the converted file with *File:Save* or with *File:Save As. . .* in case you wish to save the file under a different name.

13.3.3 Selecting the test results file

Plots cannot be shown until a matching set of test results is loaded. A matching set of test results is a test results file that contains the same variables as used in the plot(s). If the selected test results do not match (some of) the plots, these plots will be marked with a big red X.
To select a test results set, select *File:Select Test Results File. . .* and the test results file will be loaded into the variable browser. It is not possible to have multiple test results files selected at the same time.

13.3.4 Using recorder files

Usually, the recorder files used are the ones related to the selected test results file. Plots use the data from that specific test results set.
Sometimes however, it is desirable to be able to create a plot from a specific recorder file. For example, to compare the results from a certain test run to a reference run. This can be achieved by adding recorder files to the variable browser (*File:Add Recorder File. . .*).
Curves created with variables from this specific recorder file always display with the data in that specific recorder file.
Switching test result files has no effect on these curves. The variables in the curves from such a manually inserted recorder file are labeled with "[A]" (absolute).

13.3.5 Creating a new plot

To create a new plot, either select *Plot:New Plot* to create an empty plot or select *Plot:Add Plot Wizard. . .* to start the wizard that will guide you through the various needed steps to create a plot from information you provide.

13.3.6 Changing a plot

A plot is changed using the *plot properties* part of the user interface. To show the plot properties select a plot on the plot view and choose *Plot:Properties. . .*

Adding curves
Curves can be added to a plot in many ways. The easiest way is to use drag and drop. Select the variables you would like to add as curves in the variable browser and drag them to the curve editor or on the desired plot icon in the plot view. More information can be found in Section 13.4.2.

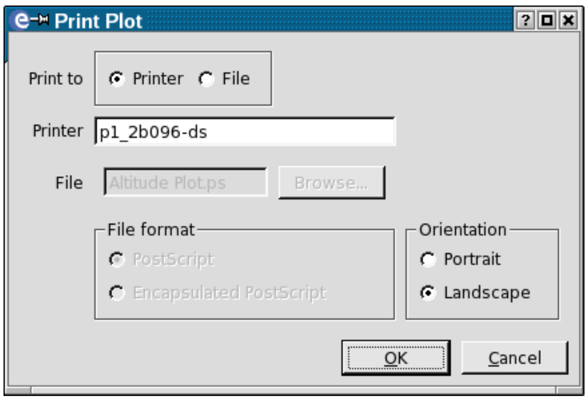
Changing curves
To change a curve or one of its properties, click on it in the curve editor. An edit field will appear depending on where you clicked. For example, clicking the variable name in one of the curves axis will show a selection box with the variables used (or recently used) in the plot.
A more detailed list of the possibilities can be found in Section 13.4.2

Removing curves
To remove a curve, select it in the curve editor and press the delete key, use the toolbar or menu (*Curve:Remove Curve*).

Changing other plot settings
General plot settings can be changed on the "General" tab page of the plot properties area. This includes settings like plot title, description, legend position etc. A more detailed list can be found in Section 13.4.1.
Settings related to the axes like scaling and range can be changed on the "Axes" tab page of the plot properties area. Detailed information can be found in Section 13.4.3

13.3.7 Showing and printing plots

After a plot has been properly set up it is shown by selecting *Plot:Show Plot* from the menu (or double-click the plot icon). A new window appears containing the plot. If gnuplot is selected as the plot back-end, the window can be closed like any other window or by selecting *Plot:Close Plot* from the menu. If PV-WAVE is the current back-end the window can only be closed by selecting *Plot:Close Plot* from the menu.
To print one or more plots, select them and choose *File:Print*. The print dialog appears.


Print Plot

Print to: [✓ Printer] File

Printer: p1_2b096-ds

File: Altitude Plot.ps Browse...

File format: [✓ PostScript] Encapsulated PostScript

Orientation: Portrait [✓ Landscape]

OK Cancel

Figure 13.3: Printing plots.

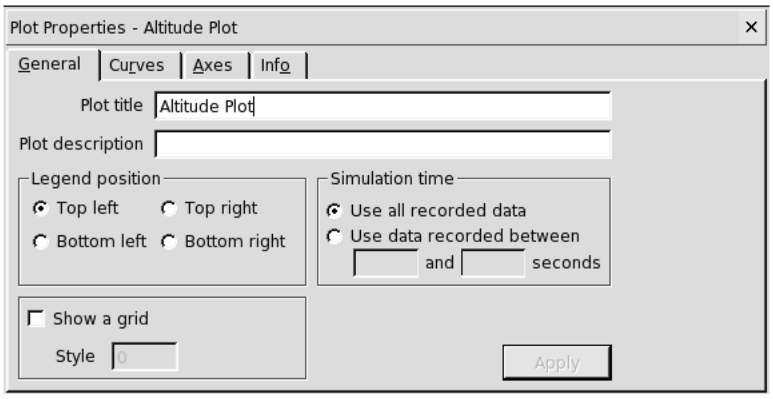
It is possible to print to the printer or to print to file(s). Printing to the printer will print each plot on a separate page, while printing to file will print each plot in a separate file.

13.4 Plot properties reference

The next three sections describe the plot properties area. This area can be used to alter the plot's properties. It is divided into three parts: general properties, the curve editor and the axes properties.

13.4.1 General plot properties

Figure 13.4 shows the tab page with the general plot properties.



Plot title: Altitude Plot
Plot description:
Legend position: [✓ Top left] Top right Bottom left Bottom right
Simulation time: [✓ Use all recorded data] Use data recorded between and seconds
☐ Show a grid
Style 0

Figure 13.4: General plot properties.

Plot title
The title of the plot is shown on the plot view as well as on the plot itself.

Plot description
This can be a more elaborate description of the plot and is shown on the plot.

Legend position
The legend is placed on the specified position.

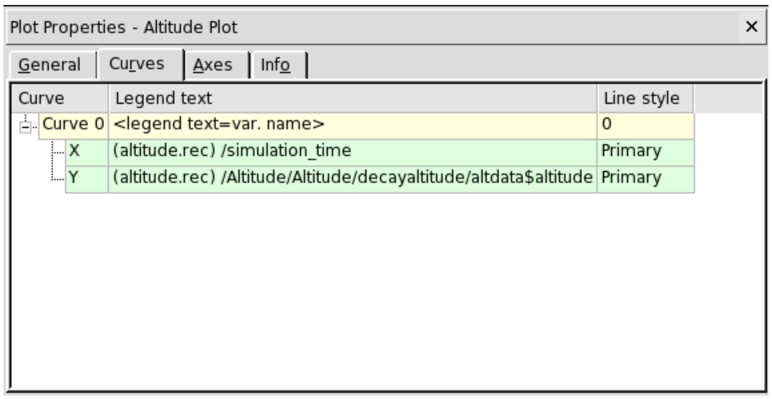
Simulation time
The simulation used in the plot can be set to either all data or to a specified time range.

Grid To display a grid check the "Show grid" option. Optionally, a grid style can be entered. The effect of the grid style depends on the back-end. In gnuplot for example, this influences the line style of the grid.

Note that the apply button must be pressed after you have made your changes.

13.4.2 Curve editor reference

The curve editor is the tool to make, change or remove curves from a plot. It displays the curves of the plot selected on the plot view.



Curve	Legend text	Line style
Curve 0	<legend text=var. name>	0
X	(altitude.rec) /simulation_time	Primary
Y	(altitude.rec) /Altitude/Altitude/decayaltitude/altdata$altitude	Primary

Figure 13.5: The curve editor.

About curves
As shown in Figure 13.5, a variable or function must be specified for the X and Y in each curve[2].
Some of the fields in the curve editor can be edited by clicking them. For example, to change the line style of a curve click on the last column of the curve's row and type in the desired style.

[2]This is different from previous versions of the Test Analyzer, where there could be only one x-axis variable or function in a plot.

Legend text
The legend text can be specified manually by typing in a legend text or it can be generated automatically. In that case, one of these formats can be chosen:

- variable name
- variable path
- variable description

Line style
The effect of the line style depends on the back-end and the output media (screen or printer). With gnuplot, for example, the decimals specify the linetype as specified in the gnuplot documentation and the hundredths specify the style. Up to nine gnuplot styles are supported. Example: the value “100” will give you the gnuplot “points” style.

Variable
The axis variable can be changed in two ways. The drop-down list contains the recently used variables in this plot and can be chosen the normal way. It is also possible to drag a variable from the variable browser and drop it on the desired axis.

Axis The axis can be set to “Primary” or “Secondary”. The primary axis is on the left for X and at the bottom for Y. The secondary axis is the right axis for X and the top axis for Y.

Adding curves
Curves can be added in many ways:

- Double click a variable in the variable browser. The selected variable is added as a curve. Initially, the variable is plotted against simulation_time so do not forget to change this if necessary.
- Drag the variables selected in the variable browser to an empty spot of the curve editor. If there is a variable with “time” or “x” in its name it is used as the x-axis variable. The curves created are all other variables plotted against this curve (or against the first variable if no such variable could be found). This is probably the easiest method.
- Select *Curves:Add Curve* from the menu. The result is the same as dragging the selected variables from the variable browser to the curve editor.

13.4.3 Axes properties

The plot’s axes can be configured with the last tabpage. Figure 13.6 shows this tabpage. On the left the axis can be selected. On the right, the settings for the current axis are shown.

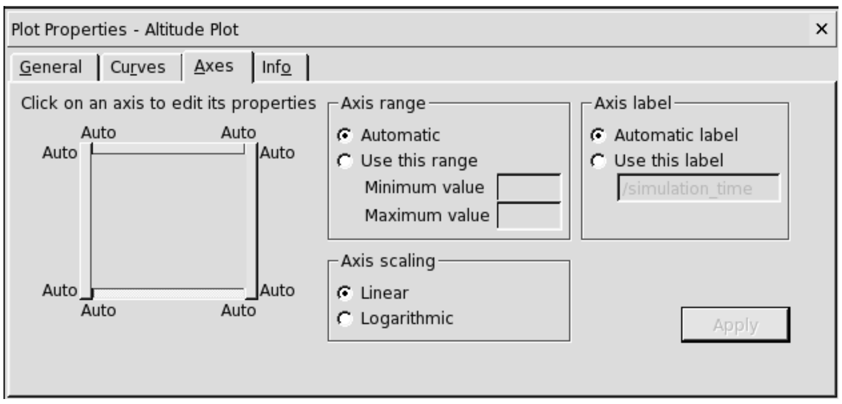
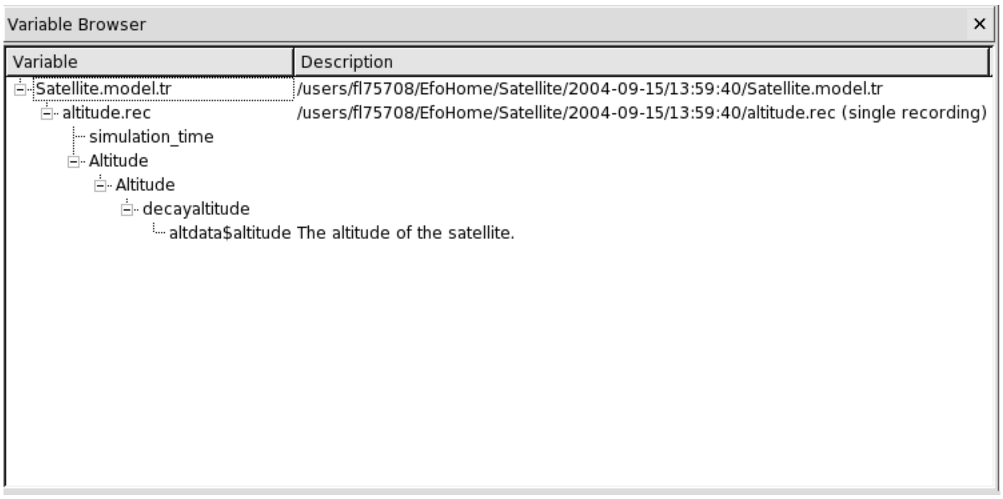


Figure 13.6: Axes properties.

The axis properties that can be set include axis range, scale and label. “Automatic axis range” calculates a default range from the data values. “Automatic axis label” creates a default label for the selected axis based on the variable names.

13.5 Variable browser reference

The variable browser displays the variables present in the currently loaded test result and recorder files. By default, all nodes are collapsed. To expand all nodes to the variable level, right-click the variable browser and choose *Expand All Nodes*.



Variable	Description
Satellite.model.tr	/users/fl75708/EfoHome/Satellite/2004-09-15/13:59:40/Satellite.model.tr
altitude.rec	/users/fl75708/EfoHome/Satellite/2004-09-15/13:59:40/altitude.rec (single recording)
simulation_time	
Altitude	
Altitude	
decayaltitude	
altdata$altitude	The altitude of the satellite.

Figure 13.7: The variable browser.

The variable browser has two columns. The first column contains the variables, the second column contains the variable descriptions.

13.6 Plot view reference

The plot view shows all defined plots. The plot view can be switched between three modes:

- Large icons
- Small icons
- List

Figure 13.8 shows the default large icons.

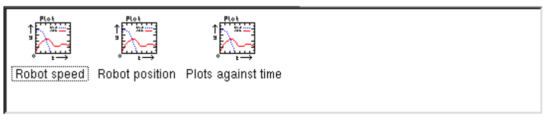


Figure 13.8: The plot view.

In small icons and list mode, the plot icon is small and the plot title is shown right of the icons instead of below them.
The difference between small icons and list mode is the order of display. In small icons mode the icons are ordered left to right while in list mode the icons are ordered top to bottom.

13.7 Menu items reference

The next sections describe each of the menus and their menu items. Some of these menu items also have a toolbar button that performs the same action. These are described in Section 13.8.

13.7.1 File menu

New Starts a new, empty .plt file.

Open...
Opens an existing .plt file. Can also be used to import old .pdf files.

Save Saves the current .plt file.

Save As...
Saves the current .plt file under the specified name.

Close Closes the current .plt file. Asks to save changes if there are unsaved changes.

Select Test Results File...
Switches the current test result set (.tr file). The variables used in the plots must be present in the new test results file, otherwise (some of) the plots will be marked as invalid. See also Section 13.3.3.

Add Recorder File...
Adds a recorder file to the current test results. See also Section 13.3.4 for more information about this feature.

Close Recorder File
Closes the recorder file selected in the variable browser. This is only possible for recorder files added with *File:Add Recorder File...*

Print...
Prints the selected plots.

Recent files
The four most recently used .plt files can be opened quickly from here.

Exit Exits the program. Asks to save changes if there are unsaved changes.

13.7.2 Edit menu

Undo
Undoes the last action if possible.

Redo
Redoes the last undone action if possible.

Cut Cuts the selected item from the document and places it on the clipboard.

Copy Copies the selected item from the document and places it on the clipboard.

Paste Inserts the item on the clipboard into the document.

13.7.3 View menu

Toggle Variable Browser...
Shows/hides the variable browser.

Large icons
Toggles the plot view to large icon mode. The icons are large, the plot title is shown below the icon and icons are initially placed right to left.

Small icons
Toggles the plot view to small icon mode. The icons are smaller, the plot title is shown next to the icon and icons are initially placed right to left.

List Toggles the plot view to list mode. The icons are small, the plot title is shown next to the icon and icons are initially placed top to bottom.

13.7.4 Plot menu

Add Plot Wizard...
Starts the wizard. The wizard allows you to create a plot step by step. All information needed to create a plot is gathered in several pages.

New Plot
Creates a new, empty plot.

Delete Plot(s)
Deletes the plots selected on the plot view.

Show Plot(s)
Shows the plots selected on the plot view.

Close Plot Window
Closes an open plot window for the selected plot. If you are using gnuplot the plot window can also be closed as usual. However, if you are using PV-WAVE you must close the plot window this way.

Print...
Prints the selected plots.

Add Selected Variables as Curves
Adds the variables selected in the variable browser as curves to the current graph. If a variable is found containing 'x' or 'time' it is used as the X-axis variable. Otherwise, the first variable is used as the X-axis.

Edit Functions
Shows the function editor dialog box for this plot. It contains all variables and user defined functions for this plot.

Properties
Shows/hides the plot properties area.

13.7.5 Curve menu

Add Curve
Adds a new curve to the current plot. See also the remarks in Section 13.4.2 about adding curves.

Remove Curve
Removes the current curve from the current plot.

13.7.6 Tools menu

Select Plot Backend
Shows a dialog in which the plot back-end can be selected. See Figure 13.9 below.

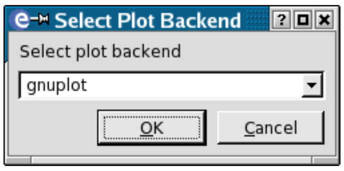


Figure 13.9: Plot back-end selection.

Plot Backend Interface
Shows the interface to the plot back-end. The interface allows you to see the responses from the plot back-end and send commands to the back-end manually. See Section 13.10.1 or Section 13.11.1 for more information.

13.7.7 Help menu

Online Help
Starts the help browser.

About EuroSim
Shows a dialog with information about EuroSim.

13.8 Toolbar reference

Many of the menu items described in the previous section are also present on the toolbar. The toolbar provides shortcuts to these menu items as toolbar buttons.
The toolbar is shown in Figure 13.10. A description of the action of each toolbar button is provided in Section 13.7. The icons on the toolbar are shown next to the menu items.

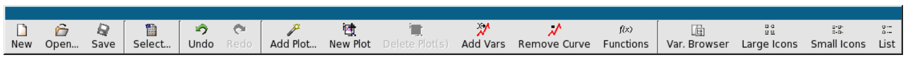


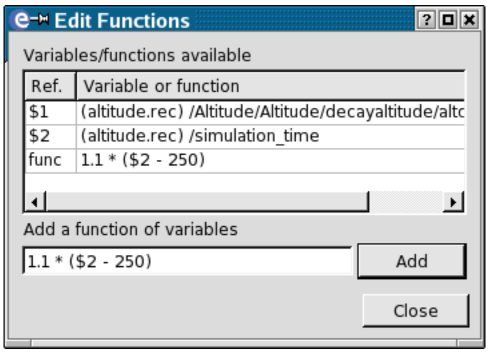
Figure 13.10: The Test Analyzer toolbar.

13.9 Using User Defined Functions

User defined functions can be specified in the function editor (see Section 13.9.1). How format and validation of these functions is handled is described in Section 13.9.2.

13.9.1 The function editor

The function editor allows you to specify a function that uses one or more of the variables of the test results. The function editor is displayed if you select *f(x)* *Plot:Edit Functions* or if you press the "Add a function of variables" button in the curve editor.



Edit Functions

Variables/functions available

Ref.	Variable or function
$1	(altitude.rec) /Altitude/Altitude/decayaltitude/altc
$2	(altitude.rec) /simulation_time
func	1.1 * ($2 - 250)

Add a function of variables

1.1 * ($2 - 250)

Figure 13.11: The function editor.

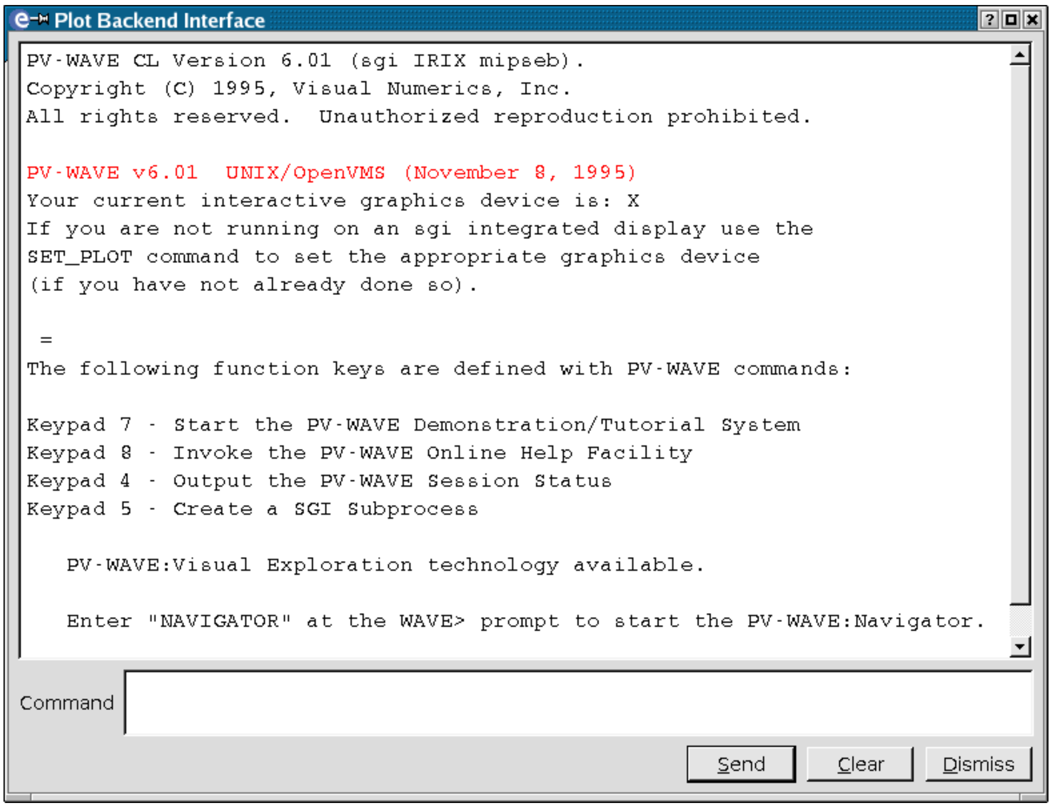
By default, the function editor displays the variables already in use by the selected plot. If a variable is required that is not yet listed, it suffices to drag and drop the variable from the variable browser onto the function editor.
To add a user defined function, type it in the edit field below the list and press the add button. User defined functions are added to the bottom of the list and are tagged as "func". They can be edited by clicking on the function. An edit field will then appear.

To use a function in a plot, drag and drop the function from the function editor to the desired axis of the desired curve in the curve editor. It is also possible to click on the variable or function field of the desired axis of the desired curve and then select the function from the list.
Note that unused functions and variables are removed between sessions. That is, if you save the .plt file and load it again unused variables and functions are not longer listed.

13.9.2 Format and Validation

The entry for the function is free format, allowing you to build functions using standard mathematical operators and expressions. To reference data from another variable (or from another user defined function), refer to the reference tag shown in front of the variable (in the "Ref." column), e.g. sin($1) will give the sine of the variable tagged as "$1" in the list. Functions are tagged as "func" in the list. Note that it is not longer possible to reference functions (i.e. it is no longer possible to nest functions).
The function typed in is sent to the plot back-end "as is". No checks are performed to see if the function is correct because each back-end has its own format for functions.
If there is an error, then the plot will not appear when *Plot:Show Plot* is requested. Common errors are recognized and the plot back-end interface window will appear. Since not all errors are recognized, it is recommended that the plot back-end interface window is kept open when plotting user defined functions (at least for the first few times), so that any errors can be quickly identified and corrected.

13.10 PV-WAVE interface



```
PV-WAVE CL Version 6.01 (sgi IRIX mipseb).
Copyright (C) 1995, Visual Numerics, Inc.
All rights reserved.  Unauthorized reproduction prohibited.

PV-WAVE v6.01  UNIX/OpenVMS (November 8, 1995)
Your current interactive graphics device is: X
If you are not running on an sgi integrated display use the
SET_PLOT command to set the appropriate graphics device
(if you have not already done so).

 =
The following function keys are defined with PV-WAVE commands:

Keypad 7 - Start the PV-WAVE Demonstration/Tutorial System
Keypad 8 - Invoke the PV-WAVE Online Help Facility
Keypad 4 - Output the PV-WAVE Session Status
Keypad 5 - Create a SGI Subprocess

   PV-WAVE:Visual Exploration technology available.

   Enter "NAVIGATOR" at the WAVE> prompt to start the PV-WAVE:Navigator.
```

Figure 13.12: The plot back-end interface window, showing PV-WAVE output.

13.10.1 PV-WAVE Operators and Functions

There are many PV-WAVE functions which can be used; the main criteria is that the function should return an array. The following are examples of valid functions (assuming that the variables tagged with $1 and $2 exist in the list of variables).

- sin($1)
- $1 ^ 2
- $1 * exp(0.1)
- $1 + (3 * $2)
- !Dtor * $2

The last example shows the use of the PV-WAVE system variable "deg to rad"; this and other possibilities are described in Section 13.10.2.
PV-WAVE has various operators and functions available, including the following:

- * / + - ^
- sin, cos, tan, sinh, cosh, etc
- alog, alog10, exp, sqrt, abs

PV-WAVE Programmers Guide (Chapter 3): describes expressions and operators
PV-WAVE Reference Volume 1 (Chapter 1): gives an overview of all the available routines; of particular relevance are the General/Special/Transcendental Mathematical Functions.
When referencing two vars within a function, e.g. "$4 - $6", the function is applied in turn to each of the values within the two datasets, e.g. the difference between the first two values, and then between the second values and so on. In the case of the two datasets having different number of recording entries, then the function is applied until the smaller set of values is exhausted.
Warning: a comparison of datasets produced by plotting *$1* and *$2* requires that */simulation_time* variables[3] from both of the source recording files are referenced, with the resulting comparison being actually an overlay of the two graphs, each using a separate time base. However, if you use a single diff function instead (e.g. *$1* - *$2*) then only one timebase is possible. This is taken from the first file that is referenced (in this example, the *$1/simulation_time* values). For this to give the intended result, the two datasets should have the same recording characteristics (i.e. have been recorded at the same frequency and be in "synchrony" (either due to the same timestamps within the recording, or because both recording files begin after the same event).

13.10.2 PV-WAVE Variables

PV-WAVE has various system variables available, of which the following may be useful:

- !Pi: The floating-point value of *pi*: 3.14159
- !DPi: Contains the double-precision value of *pi*: 3.1415927
- !Dtor: Contains the conversion factor to convert degrees to radians. The value is *pi*/180, which is approximately 0.0174533
- !Radeg: A floating-point value for converting radians to degrees. The value is 180/*pi* or approximately 57.2958

PV-WAVE Reference Volume 2 (Chapter 4): gives an overview of all the available system variables (although the majority are concerned with plot appearances/defaults and are not relevant for function definitions).

[3]It is assumed that `simulation_time` is used for the x axis variable, but it could be some other variable of course.

13.10.3 Accessing recorded data

After a plot has been activated, the plot back-end interface window will show the exact commands sent to PV-WAVE (in blue). If we inspect this output, we can see that the variables used in our plot ($1, $2, etc.) are available as V1, V2, etc. The dollar sign ($) of the variable reference is replaced with a "V". We can access these variables in PV-WAVE as usual. For example, to check the number of data values for $1 we can give the command:

```
info, V1
V1 DOUBLE = Array(307)
```

Which means that V1 is an array of 307 elements.
To actually see the values in the array we could issue a print command:

```
print, V1
0.0029616649 0.0059233298 0.0088849947 0.011846660
0.014808325 0.00092749497 0.0038891599 0.0068508248
0.0098124897 0.012774155 0.015670285 0.0018549899
.......
```

13.10.4 Examples of using PV-WAVE commands directly

PV-WAVE provides many options for presenting/filtering data. These can be used by typing the commands in the back-end interface dialog window and sending them to the PV-WAVE process.
Some examples of the use of these commands on recorded data are presented below.

13.10.4.1 Creating a table

To create a table of the data from a recorder file, the following commands could be used:

```
simtime = V1
temp1 = V2
temp2 = V5
temp3 = V6
tempTable = build_table("simtime, temp1, temp2, temp3")
```

Now, to select and display a subset of the data the following commands can be used:

```
subsetTable = query_table(tempTable,
  " * Where simtime > 10.0 and simtime < 12.0")
print ,"time celltmp[1][1] celltmp[1][2] celltmp[1][3]"
for i=0, N_ELEMENTS(subsetTable)-1 do begin PRINT, subsetTable(i)
```

This will result in output similar to:

```
time celltmp[1][1] celltmp[1][2] celltmp[1][3]
{ 10.005000 193.298 169.990 260.438}
{ 10.015000 193.298 169.990 260.438}
......
```

To export the selected data, and store it in a file (as ASCII), use the following command:

```
status = DC_WRITE_FIXED('table.dat',subsetTable.simtime,
subsetTable.temp1,subsetTable.temp2,subsetTable.temp3,/Col )
```

13.10.4.2 Data analysis

On the recorded data, analysis functions such as a Fast Fourier Transform (FFT) can be performed. An example would be:

```
xd = simtime
yd = temp1
n_sample = N_ELEMENTS(xd)
samp_rate = (n_sample-1)/(xd(n_sample-1) - xd(0))
x = FINDGEN(n_sample) - (n_sample/2.)
x_ind = WHERE(x GE 0)
x(x_ind) = x(x_ind)+1.
x_freq = x * samp_rate/ FLOAT(n_sample)
y_proc = ABS(FFT(yd, -1))
PLOT, x_freq, SHIFT(y_proc, n_sample/2.)
```

A FFT plot should then appear. Plots generated with the `plot` command can be removed again by using the command `wdelete,0` (for plot number 0)
Also, various statistical analysis functions are available through PV-WAVE. For example:

```
print, "min= ", min(yd)
print, "max= ", max(yd)
print, "mean= ", avg(yd)
print, "median= ", median(yd)
print, "std dev= ", stdev(yd)
```

13.10.5 User defined functions

It is possible to define user defined functions which can later be used interactively in the dialog box which shows the interface with the plot back-end. To create a new user defined function you must first create a file containing the commands. From the Test Analyzer menu *Tools:Shell…* a shell window can be opened where you can create a file using your favorite editor. The filename should be the name of the function and the filename extension should be ‘pro’, e.g. user_func.pro. Type the PV-WAVE commands in the file and save it. In the Test Analyzer select *Tools:Plot Backend Interface…* A dialog box appears where you then can enter your command in the Command box as follows: “.run user_func”. Click Send to execute the command.

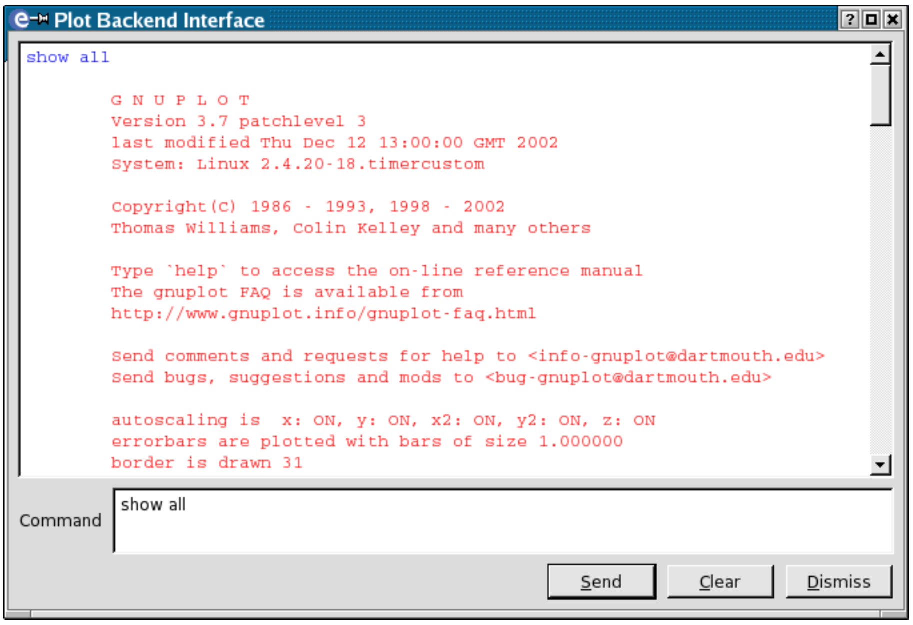
13.10.6 PV-WAVE help

This can be accessed from the back-end interface dialog window by sending the command help.

13.10.7 The PV-WAVE process

As soon as the current plot back-end is set to PV-WAVE, an attempt is made to start PV-WAVE. Depending on the number of PV-WAVE licenses available in the local environment however, this might not succeed. If the start-up fails, then the user’s request for a license is placed in a queue. All the Test Analyzer edit functions are still available however and the user can make/edit plot definitions as required: the only difference is that the “activate” (display graphical plot) request will not be immediately executed.
If the Test Analyzer appears unresponsive to requests to display a plot, then the back-end interface window should be checked for this situation and/or other error messages.

13.11 gnuplot interface



```
show all

        G N U P L O T
        Version 3.7 patchlevel 3
        last modified Thu Dec 12 13:00:00 GMT 2002
        System: Linux 2.4.20-18.timercustom

        Copyright(C) 1986 - 1993, 1998 - 2002
        Thomas Williams, Colin Kelley and many others

        Type `help` to access the on-line reference manual
        The gnuplot FAQ is available from
        http://www.gnuplot.info/gnuplot-faq.html

        Send comments and requests for help to <info-gnuplot@dartmouth.edu>
        Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>

        autoscaling is  x: ON, y: ON, x2: ON, y2: ON, z: ON
        errorbars are plotted with bars of size 1.000000
        border is drawn 31
```

```
show all
```

Figure 13.13: The plot back-end interface window, showing gnuplot output.

13.11.1 gnuplot operators and functions

According to the gnuplot documentation, the expressions accepted by gnuplot can be any mathematical expression that is valid in C, FORTRAN, Pascal or BASIC. The precedence of operators is the same as in the C programming language.
The functions supported by gnuplot are about the same as those present in the UNIX math library. A complete list is available in the gnuplot documentation. Examples:

- sin($1)
- log10($3)
- $1**2 *[this means $1 squared]*
- $1 * exp(0.1)
- $1 + (3 * $2)

13.11.2 Accessing recorded data

Showing a plot causes a temporary file to be written containing the variables used in the plot. This file will be deleted when the Test Analyzer is closed or when the back-end is set to something else than gnuplot. In the meantime, the data in this file remains accessible.
The name of the data file can be obtained from the plot back-end interface window. After showing a plot, the name of the datafile is shown on the line containing the plot command, for example:

```
plot "/var/tmp/gnuAAAa0Y093" using ($1):(1.1 * ( $2 - 250 )) axes

x1y1 title "just a plot'' with lines lt 0
```

The name of the file is shown in bold. The data can be accessed using gnuplot's **using** command, as shown in the plot command above. See the gnuplot documentation for more information.

13.11.3 gnuplot help

The gnuplot help interface can be accessed by sending the “help” command from the back-end interface window. Note that you should press enter a few times to leave help mode.

Chapter 14

Batch utility reference

14.1 Introduction

This chapter provides details on the batch utility[1]. The utility uses perl as the scripting engine. Various perl modules have been created that provide an interface to existing EuroSim libraries. This means that a batch script is no more than an ordinary perl script using EuroSim modules.
The main reason to choose perl as the batch utility engine is that it is the ultimate glue language. The EuroSim modules can be combined with the built-in features of perl itself or with one of the many perl modules which are freely available on the internet. Check out CPAN (Comprehensive Perl Archive Network) at www.cpan.org for a complete overview of all available perl modules.
There is an interactive shell which can be used to type commands directly on the command line to start and manipulate simulators. This tool has been implemented in perl using the EuroSim modules and a few other helper modules for the command line interaction.
Section 14.2 describes the conversion utility for people using the event-probe tool. Section 14.3 shows you how to use the interactive batch shell. Section 14.4 explains all EuroSim modules. Section 14.5 shows you how to extend the batch utility to integrate it in a larger system. Section 14.6 contains a simple example script.

14.2 Conversion utility for event-probe users

Event-probe is an unsupported batch utility program which was meant to be used for internal testing only.
In order to facilitate the users of this tool to convert to the new batch facilities a conversion tool has been supplied. This tool is called `probe2esh`. To convert an existing event-probe script use the following command:

```
probe2esh < probe_script > perl_script
```

For more information read the manual page *probe2esh(1)*.

14.3 Starting the interactive batch shell

The EuroSim command line shell is started by running the `esimsh` command. The `esim>` prompt appears and you can start typing commands. The shell has various forms of completion. Typing TAB once will show you a complete list of available commands. Each command is in fact a perl function provided by the EuroSim modules. Read the manual pages for detailed information on arguments and return values.
You can save the commands by using the built-in logging function. This function is started by calling `log_open` “*perl-script*”. All commands entered after this are written to the file called *perl-script*. This

[1]Not supported on the Windows NT platform.

file can then be used as a starting point for further non-interactive runs. To stop logging commands you call `log_close`.
When you start a simulation in interactive mode (the default when starting esimsh) an xterm window is started to show the journal messages.

14.4 Batch utility modules

The batch utility consists of one module for each object. This follows the perl object-oriented design features. It means that given an object you can call methods in the following manner:

```
$object->method($arg1, $arg2);
```

There is one module which forms an exception to this rule for convenience reasons when using the interactive shell: `EuroSim::Session`. All functions (methods) can be called directly without the object reference. This is done to reduce typing in the interactive shell. Each function uses the current session. This works fine as long as you only have one session. If you want to manage multiple sessions in parallel within one script you must use the full notation.

14.4.1 EuroSim::Session module

This is the central module used to run simulations. It supports the complete client/server protocol with the running simulator executable. For each command you can send to the simulator there is a function. For each message sent from the simulator to the application you can install a callback. You can also wait synchronously for any message. The messages and responses are documented in detail in Appendix J. The idea behind this module is that it is a replacement for the simulation controller. It can fully automate anything you can do with those tools.
To start a simulator all you need to do is:

```
$s = new EuroSim::Session("some.sim");
$s->realtime(1);
$s->init;
```

This command will use the information defined in the simulation definition file to start the simulator. The `realtime` flag results in a real-time run of the simulator.
As you can see you pass similar information to the function call as needed by the simulation controller. In the simulation controller you open a simulation definition file and then you select whether or not you want to run real-time. Then you hit the init button, which launches the simulator. The simulation controller automatically connects to the simulator, just like the `init` function does. This function also sets up a number of callback functions for incoming events. The information carried by each event is stored in the session structure. The user can at any moment print the contents of this structure by calling `print_session_parameters`.
To install a new handler for an event you call the function `event_addhandler` with the name of the event you want to handle and the callback to call for that event. You can install more than one handler for each event. Handlers are called in the order they were installed. The name of the event is the same as the name of the enumeration identifier, e.g. `rtExecuting`. To remove the handler, call `event_removehandler` with the same parameters.
Each callback receives the following parameters:

1. Session object, reference to the session hash (see Section 14.4.1.1)
2. Name of the event (name of the enumeration identifier)
3. Simulation time (sec)
4. Simulation time (nsec)
5. Wallclock time (sec)

6. Wallclock time (nsec)
7. Parameters (event specific)

Example:

```
sub cb_standby
{
 my ($session, $event_name, $simtime_sec, $simtime_nsec,
    $wallclock_sec, $wallclock_nsec) = @_;
 print "going to standby at $wallclock_sec\n";
}
$session->event_addhandler("rtStandby", \&cb_standby);

# or a bit more compact
$session->event_addhandler("rtExecuting",
                         sub { print "going to executing at $_[4]\n"; });
```

It is possible to synchronously wait for an event you expect. In this case you call `wait_event` with the name of the event (same name as used to install a handler) and an optional time-out.
To synchronously wait for some time to pass, you can call `wait_time`. This function takes the number of seconds you want to wait as an argument.
A complete overview of all functions provided by this module can be found in the manual page *EuroSim::Session(3).*

14.4.1.1 Session data structure reference

The Session object is a hash table with the following fields:

MDL Hash table of loaded MDL files. Each hash key is the name of a loaded MDL file. The hash value is a `EuroSim::MDL` object. MDL files are loaded at start-up when a .sim file is loaded or during run-time when extra MDL files are loaded. Extra files can be loaded by the built-in event handler for event `maNewMission` or by manually adding MDL files with `new_scenario`.

clientname
The name under which this session is known to the simulator. The value is set with the function `clientname`.

conn `EuroSim::Conn` object. Low level connection object.

cwd Current working directory of the simulator. The value is set by the built-in event handler for event `maCurrentWorkingDir`.

am_cycle_time
Action Manager cycle time in secs. The value is set by the built-in event handler for event `maCurrentCycleTime`.

dict Data dictionary file name. The value is set by the built-in event handler for event `maCurrentDict`.

eventlist
List of events present in the schedule. The value is set by the built-in event handler for the following events: `scEventListStart`, `scEventInfo`, `scEventListEnd`. The `eventlist` is an array of hash tables. Each table consists of three elements:

name The name of the event.

state The scheduler state for which it is defined.

is_standard
Flag indicating that it is a standard event, i.e. predefined by EuroSim.

handler Event handler table.

sim_hostname
Simulation host name. The value is set with the function `sim_hostname`.

initconds
Initial condition files. The value is set by the built-in event handler for event `maCurrentInitconds`.

journal `EuroSim::Journal` object.

logwindow
`EuroSim::Window` object. Used to display simulation messages in interactive mode.

monitored_vars
Table of monitored variables.

output_dir
Result directory used in current simulation run. The value is set by the built-in event handler for event `maCurrentResultDir`.

prefcon Connection number.

realtime
Realtime mode. `1` is real-time, `0` is non-realtime. The value is set by the built-in event handler for event `scGoRT`.

recording
Flag indicating that recording is enabled or not. `1` means enabled. `0` means disabled. The value is set by the built-in event handler for event `maRecording`.

recording_bandwidth
Recorder bandwidth in bytes/second. The value is set by the built-in event handler for event `maRecordingBandwidth`.

schedule
Schedule file name. The value is defined in the simulation definition file.

simdef Simulation definition handle to a `EuroSim::SimDef` object.

sim_time
The simulation time (as seen by the running simulator). The value is set by the built-in event handler for event `dtHeartBeat`.

speed The clock acceleration factor achieved by the simulator. Values larger than `1` indicate faster than real-time. Values smaller than `1` indicate slower than real-time. The value is set by the built-in event handler for event `scSpeed`.

state Simulator state. Can be: unconfigured, initialising, standby, executing, exiting. The value is set by the built-in event handler for the following events: `rtUnconfigured`, `rtInitialising`, `rtStandby`, `rtExecuting` and `rtExiting`.

stimulus_bandwidth
Stimulus bandwidth in bytes/second. The value is set by the built-in event handler for event `maStimulatorBandwidth`.

tasklist List of tasks present in the schedule. The value is set by the built-in event handlers for the events `scTaskListStart`, `scTaskStart`, `scTaskEntry`, `scTaskEnd` and `scTaskListend`. The field tasklist is a hash table. Each key in the hash table is the name of a task (e.g. `$session->tasklist->tasknam`
Each task consists of a number of entry points and a flag called disable. The disable flag is set by the built-in event handler of `scTaskDisable`. The entry points are stored in an array. Each array element is a hash table consisting of three fields:

name The name of the entry point.

breakpoint
Flag indicating that a breakpoint has been set on this entry point. The value is set by the built-in event handler for event `scSetBrk`.

trace Flag indicating that this entry point is being traced. The value is set by the built-in event handler for event `scSetTrc`.

time_mode
The time mode can be relative or absolute (UTC). Relative is `0` and absolute is `1`. The value is set by the built-in event handler for event `maCurrentTimeMode`.

user_defined_journal
User defined journal filename. This journal filename overrides the default journal filename. The value is set with the function `journal`.

user_defined_outputdir
User defined output directory path. This directory path overrides the default output directory path. The value is set with the function `outputdir`.

wallclock_time
The wallclock time (as seen by the running simulator). The value is set by the built-in event handler for event `dtHeartBeat`.

wallclock_boundary
The wallclock boundary time to be used for timed state transitions. If you add an integer number of times the main_cycle time to this value it will produce a valid state transition boundary time.

simtime_boundary
The simulation time boundary to be used for timed state transitions. If you add an integer number of times the main_cycle time to this value it will produce a valid state transition boundary time.

main_cycle
The main cycle time of the current schedule. It can be used to calculate valid boundary times for timed state transitions.

watcher
`Event::io` object. Used to process incoming events.

where Current breakpoint. The value is set by the built-in event handlers for the following events: `scWhereListStart`, `scWhereEntry`, `scWhereListEnd`. It is cleared by the following events: `scStepTsk` and `scContinue`. The value is an array of value pairs stored in an array. The first value in the array is the task name and the second is the entry number. For example:

```
print "task: $s->{where}->[0][0]\n";
print "entry_nr: $s->{where}->[0][1]\n";
```

write_access
Flag to indicate whether this client is allowed to change variable values in the simulator. The value is set by the built-in event handler for event `maDenyWriteAccess`.

14.4.1.2 Monitoring variables

In order to monitor variables you must call the function `monitor_add` with the variable you want to monitor. The variable parameter is in the form of a valid EuroSim data dictionary path. This function will add the variable to the list of variables monitored in EuroSim. The value of each variable will be updated with a frequency of 2 Hz if they change. If there is no change, no update is sent.

The values of the variables are stored in the `monitored_vars` hash array of the session hash array. To access the value of a variable use the following expression: `$s->{monitored_vars}->{var_path}`.
To stop monitoring a variable you must call the function `monitor_delete` with the variable you want to stop monitoring.
If you only want to get the value of a variable once, it is better to call the function `monitor_get`. This function retrieves the value of the variable immediately from the simulator, but only once. The value of the variable is in the return value.

14.4.1.3 Modifying variables

If you want to change the value of a variable in the simulator you can simply call `monitor_set` with the name and value of the variable. The value will be set as soon as possible in the simulator.

14.4.2 EuroSim::MDL module

This is a wrapper module for the EuroSim Script functions. These functions manipulate MDL files and actions.
The following (sets of) functions are available:

- read MDL file
- write MDL file
- add actions to the MDL file
- delete actions from the MDL file
- utility functions to ease the creation of new actions

There are four functions to generate action text:

script_action
  create a generic action script

monitor_action
  create a monitor action script

recorder_action
  create a recorder action script

stimulus_action
  create a stimulus action script

A complete overview of all functions provided by this module can be found in the manual page *EuroSim::MDL(3)*.

14.4.3 EuroSim::Dict module

This is a wrapper module for the EuroSim data dictionary functions. You can open and close EuroSim data dictionary files. You can get and set individual values of variables. This is used in conjunction with the initial condition module.
This module is also used for command line completion in interactive mode to complete the path of data dictionary variables.
A complete overview of all functions provided by this module can be found in the manual page *EuroSim::Dict(3)*.

14.4.4 EuroSim::InitCond module

This module offers reading and writing of initial condition files. You can also use it to combine multiple initial condition files into one file. In conjunction with the `EuroSim::Dict` module it is possible to set variables to specific values, and then save them in an initial condition file.
The following steps must be taken to change values in an initial condition file:

1. Load a data dictionary file.
2. Load one or more initial condition files into that data dictionary
3. Set one or more values of variables to their initial values.
4. Save the initial condition file with the new values.

This initial condition file can be used in a new simulation run, or it can be loaded into an already running simulator. In order to load it into a running simulator, the simulator must be in standby state, or it can be used for reinitialization.
A complete overview of all functions provided by this module can be found in the manual page *EuroSim::InitCond(3)*.
Example:

```
# load a data dictionary
$dict = EuroSim::Dict::open("test.dict");

# load initial values into that dictionary
$initcond = EuroSim::InitCond::read("test.init", $dict);

# get an initial condition value
$value = $dict->var_value_get("/test/var1");

# set an initial condition value
$dict->var_value_set("/test/var2", 3.1415);

# save the new initial condition file in ASCII format
$initcond->write("test2.init", 0);
```

14.4.5 EuroSim::Link module

This module wraps the EuroSim TM/TC Link library (see Chapter 17). You can create a TM/TC link and connect to a running simulator with `link_open` and `link_connect`. Then you can read and write to the link from perl using the functions `link_read` and `link_write`. When you are finished you can call `link_close`.
A complete overview of all functions provided by this module can be found in the manual page *EuroSim::Link(3)*.

14.4.6 EuroSim::Conn module

This is the low-level module used to send and receive events (messages) from/to a running simulator. All of these functions are used internally by the `EuroSim::Session` module.
To print a list of all events use `print_event_list`. This function prints a list of all events, their internal event number and their arguments.
A complete overview of all functions provided by this module can be found in the manual page *EuroSim::Conn(3)*.

14.5 Extending the batch utility

The batch utility is based on the Event module. This perl module provides a framework where you can integrate various systems with each other. The client-server connection with the simulator sends packets to its clients (such as the batch utility). These packets are handled by a callback (watcher in Event module terminology). The Event module is used to perform the mapping between incoming data on a socket to the central event dispatching function of the `EuroSim::Session` module. Also the wait functions are implemented by using the timer watcher.
The interactive EuroSim shell is implemented using this module. The input is processed by the package `Term::ReadLine::Gnu`. This package reads commands from stdin. The readline input function is hooked into the Event framework using an io watcher. The EuroSim connection is handled by another `Event::io` watcher. This enables the interactive shell to stay interactive. It reads simultaneously from the standard input and from the EuroSim socket. This mechanism can be extended to your needs. For a complete reference check out the *Event(3)* manual page.

14.6 Example

The following example is a complete script which performs one simulation run. Some event handlers are installed as well as some monitors.

Batch script example

```
#!/usr/bin/perl
# This is an example perl script using the EuroSim bindings
# to automate a simulation run.
# Import all modules.
use EuroSim ':all';
use EuroSim::InitCond ':all';
use EuroSim::Session ':all';
use EuroSim::Link ':all';
use EuroSim::Conn ':all';
use EuroSim::MDL ':all';

# Load the simulation definition file.
$s = new EuroSim::Session("some.sim");

# Set to real-time.
$s->realtime(1);

# Define a callback to be called when standby state is reached.
sub cb_standby
{
   my ($session, $event_name, $simtime_sec, $simtime_nsec,
      $wallclock_sec, $wallclock_nsec) = @_;
   print "going to standby at $wallclock_sec\n";
}

# Install the callback.
$s->event_addhandler("rtStandby", \&cb_standby);

# The same thing but then a bit more compact.
# Isn't perl wonderful :-)
$s->event_addhandler("rtExecuting",
                sub { print "going to executing at $_[4]\n"; });

# Start the simulation run.
$s->init;
```

```
# Wait for standby state.
$s->wait_event("rtStandby");

# Add a monitor for variable "/test/var1".
# Note that the $ sign in fortran variables must be escaped.
$var = "/test/var1";
$s->monitor_add($var);

# Wait one second. This should be more than enough for the 2Hz
# update to take place.
$s->wait_time(1);

# Print the value of the monitored variable.
print "The value of $var is $s->{monitored_vars}->{$var}\n";

# Trigger an event "my_event".
$s->raise_event("my_event");

# Trigger another event at some time in the future. In this
# case at simulation time 5.025 s.
$s->raise_event_at_simtime("another_event", 5, 25000000);

# Trigger an action in an MDL script.
$s->action_execute("some_loaded.mdl", "inject a failure");

# Go to executing state.
$s->go;

# Wait for the state transition to executing state.
$s->wait_event("rtExecuting");

# Schedule a state transition to standby state at simulation
# time 1000.0 s.
$s->freeze_at_simtime(1000, 0);

# Wait for the state transition to standby state.
$s->wait_event("rtStandby");

# Stop the simulation.
$s->stop;

# Wait until the connection with the simulator is shut down.
$s->wait_event("evShutdown");

# Quit the script.
$s->quit;
```

14.7 Useful command line utilities

There are two EuroSim command line utilities that can be very useful in combination with the batch utility. They are briefly described in the following subsections.

14.7.1 efoList

The `efoList` command line utility shows a list of currently running simulators. See the ICD document or the manual page *efoList(1)* for information on the command line options that can be passed to `efoList`.

14.7.2 efoKill

The `efoKill` command line utility lets you terminate a running simulator. See the ICD document or the manual page *efoKill(1)* for information on the command line options that can be passed to `efoKill`.

Chapter 15

Simulation Model Portability interface reference

15.1 Introduction

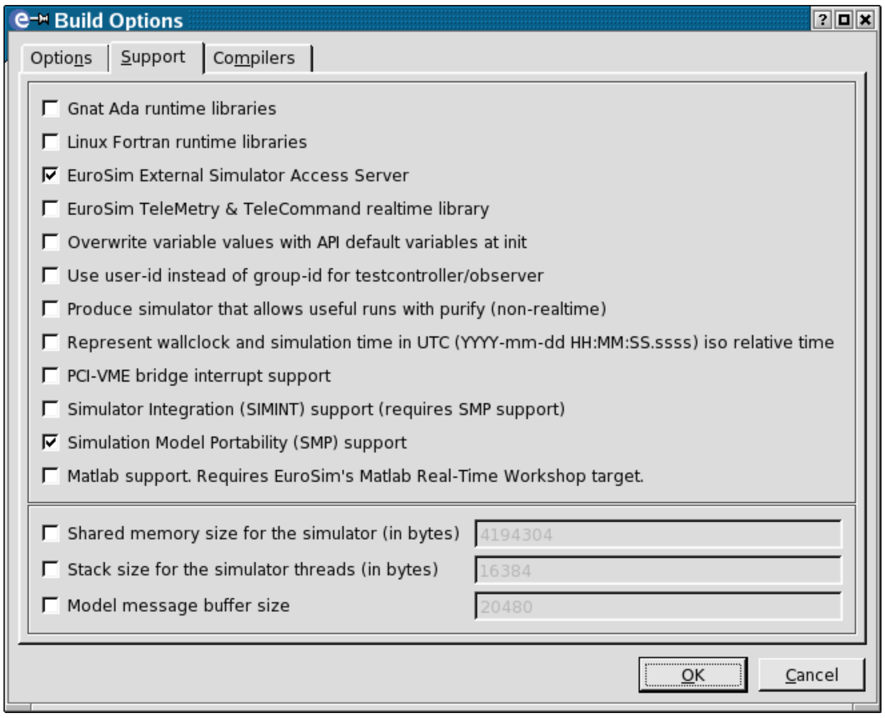
The purpose of the Simulation Model Portability SMP standard is to promote portability of models among different simulation environments and re-use of simulation models.
EuroSim complies to the SMP standard by offering the complete SMP API and by easily importing existing models into the EuroSim tooling. Information on using the SMP functions can be found in the handbook [SMP03].

15.2 Import and build procedure

Importing SMP compliant models is done by simply creating a normal model file containing all the C source files of the SMP simulator. The model manager source must provide two functions:

- `void ManagerPublishInterfaces(void)`
- `void ManagerInitialise(void)`

The first function is responsible for publishing the interfaces of the models. It calls the publish functions for all the models it manages. The second function is responsible for the initialisation of the models.
The user has to enable the Simulation Model Portability support option in the Model Editor build options dialog.



- [ ] Gnat Ada runtime libraries
- [ ] Linux Fortran runtime libraries
- [x] EuroSim External Simulator Access Server
- [ ] EuroSim TeleMetry & TeleCommand realtime library
- [ ] Overwrite variable values with API default variables at init
- [ ] Use user-id instead of group-id for testcontroller/observer
- [ ] Produce simulator that allows useful runs with purify (non-realtime)
- [ ] Represent wallclock and simulation time in UTC (YYYY-mm-dd HH:MM:SS.ssss) iso relative time
- [ ] PCI-VME bridge interrupt support
- [ ] Simulator Integration (SIMINT) support (requires SMP support)
- [x] Simulation Model Portability (SMP) support
- [ ] Matlab support. Requires EuroSim's Matlab Real-Time Workshop target.

☐ Shared memory size for the simulator (in bytes)	4194304
☐ Stack size for the simulator threads (in bytes)	16384
☐ Model message buffer size	20480

Figure 15.1: Model Editor Build Options dialog box with SMP support enabled.

After this the user only has to run the *Build All* command to compile the model source into a runnable simulator. The command also generates a schedule file from the SMP scheduling functions called during the initialization function. This file can be edited by the EuroSim schedule editor. The generated file is placed in the directory where all generated files are placed. This directory is derived from the model file name and is called *model.OS*, where *OS* can be IRIX 64, IRIX, Linux or WINNT. This directory and its contents are removed when the user runs the *Cleanup* command. It is therefore wise to save the file into another directory when changes have been made. The best directory for this would be the directory where you have stored the model file.

15.3 Example code

In the EuroSim installation directory you can find under the `src` directory, a directory with SMP example project called `SMP`. This is a very simple test simulator which shows you a working example. Section Section 15.7 describes how to create a simple SMP model based on the C++ programming language.

15.4 Limitations

Currently the scheduling calls performed in the SMP code are translated to EuroSim equivalents in the schedule file when they are called during initialization and not during a run. Because of this, the parameters of a service are ignored and cannot be used. It is also not possible to register services with a zero cycle time. These limitations are recorded as EFO-SPR-2635 in the EuroSim SPR database.
Published data and services are placed under the `SMP` org node in the EuroSim data dictionary.

15.5 Compliance Matrix

SMP function	**Supported**
SMIAddCallbackRequest	Y

Table 15.1: SMP 1.4 Compliance Matrix

SMP function	**Supported**
SMICreateParameters	Y
SMIDuplicateParameters	Y
SMIEnvAddServiceToSchedule	Y
SMIEnvGetSimulatedTime	Y
SMIEnvReadAsciiObjectData	Y
SMIEnvWriteAsciiObjectDetails	Y
SMIEnvWriteMessage	Y
SMIFreeParameters	Y
SMIGetArrayDetails	N
SMIGetCallbackCount	Y
SMIGetCallbackID	Y
SMIGetDataDetails	Y
SMIGetDataID	Y
SMIGetDataValue	Y
SMIGetObjectCount	Y
SMIGetObjectDetails	Y
SMIGetObjectID	Y
SMIGetParameterDetails	Y
SMIGetParameterID	Y
SMIGetRootObjectCount	Y
SMIGetRootObjectID	Y
SMIGetRootObjectIDs	Y
SMIGetServiceDetails	Y
SMIGetServiceID	Y
SMIGetSizeOfType	Y
SMIGetStructureDetails	N
SMIGetSubObjectCount	Y
SMIGetSubObjectID	Y
SMIGetSubObjectIDs	Y
SMIGetTypeSize	N
SMIInitialise	Y
SMIInvokeCallback	Y
SMIInvokeService	Y
SMIIsValidBaseOrUserTypeID	N
SMIIsValidCallbackID	Y
SMIIsValidDataID	Y
SMIIsValidObjectID	Y
SMIIsValidParameterID	Y

Table 15.1: SMP 1.4 Compliance Matrix

SMP function	Supported
SMIIsValidServiceID	Y
SMIIsValidTypeID	Y
SMIPublishCallback	Y
SMIPublishData	Y
SMIPublishObject	Y
SMIPublishParameter	Y
SMIPublishService	Y
SMIPublishSubObject	Y
SMIRegisterArrayType	N
SMIRegisterStructureElement	N
SMISetDataValue	Y
SMITransferData	Y

Table 15.1: SMP 1.4 Compliance Matrix

15.6 EuroSim extensions

The SMP implementation in EuroSim offers a number of extra functions which allow the user to add descriptions and units to objects and data items. These descriptions and units are shown in the data dictionary browsers of EuroSim. The unit is also shown in monitors after the value. Normally this information is entered in the Model Editor, but SMP bypasses that editor. These functions add back the EuroSim capability to annotate objects, variables and entrypoints.

SMP function	Description
SMIExtSetObjectDescription	Attach a textual description to an object or sub-object
SMIExtSetDataDescription	Attach a textual description to a data item
SMIExtSetDataUnit	Specify a unit for a data item
SMIExtSetServiceDescription	Attach a textual description to a service

Table 15.2: EuroSim SMP extensions

15.6.1 SMIExtSetObjectDescription

15.6.1.1 Functional description

Attach a textual description to an object or sub-object.

15.6.1.2 Formal description

```
Boolean_t SMIExtSetObjectDescription(const ObjectID_t ObjectID,
                                     const char *description);
```

Parameter	**Description**
ObjectID	Identifies the object of which the description is set
description	Textual description
Return Value	TRUE if the operation succeeded, otherwise FALSE

Table 15.3: Parameters

15.6.2 SMIExtSetDataDescription

15.6.2.1 Functional description

Attach a textual description to a data item.

15.6.2.2 Formal description

```
Boolean_t SMIExtSetDataDescription(const ObjectID_t ObjectID,
                                   const DataID_t DataID,
                                   const char *description);
```

Parameter	Description
ObjectID	Identifies the object to which the data item belongs
DataID	Identifies the data item of which the description is set
description	Textual description
Return Value	TRUE if the operation succeeded, otherwise FALSE

Table 15.4: Parameters

15.6.3 SMIExtSetDataUnit

15.6.3.1 Functional description

Attach a textual unit specification to a data item.

15.6.3.2 Formal description

```
Boolean_t SMIExtSetDataUnit(const ObjectID_t ObjectID,
                            const DataID_t DataID,
                            const char *unit);
```

Parameter	Description
ObjectID	Identifies the object to which the data item belongs
DataID	Identifies the data item of which the unit is set
unit	Textual representation of the unit
Return Value	TRUE if the operation succeeded, otherwise FALSE

Table 15.5: Parameters

15.6.4 SMIExtSetServiceDescription

15.6.4.1 Functional description

Attach a textual description to a service.

15.6.4.2 Formal description

```
Boolean_t SMIExtSetServiceDescription(const ObjectID_t ObjectID,
                                      const ServiceID_t ServiceID,
                                      const char *description);
```

Parameter	Description
ObjectID	Identifies the object to which the service belongs
DataID	Identifies the service of which the description is set
description	Textual description
Return Value	TRUE if the operation succeeded, otherwise FALSE

Table 15.6: Parameters

15.7 Step by step example

This section describes a simple example of how to integrate a C++ sub-model using SMP.
Suppose we have the following C++ source code file `simple.cpp`:

```
#include <esim.h>
#include <smi.h>

class Simple
{
public:
  Simple();
  bool initialize(const char *name);
  bool update(Parameter_t *parameters);
private:
  Integer32_t foo;
};
```

The constructor of the class initializes the member `foo` to zero:

```
Simple::Simple() :
  foo(0)
{
}
```

The `update` method of our class:

```
bool Simple::update(Parameter_t *parameters)
{
  foo++;
  return true;
}
```

The `update` method cannot be registered directly, as SMP is not C++ aware. We introduce a wrapper function so we can retrieve the original `this` pointer and then call the actual `update` method on the instance of our class:

```
Boolean_t update_wrapper(Data_t *pObject, Parameter_t *parameters)
{
  Simple *simple = static_cast<Simple*>(pObject);
  return simple->update(parameters);
}
```

Once an instance of the class has been created, we can call the `initialize` method. It will publish the object instance to the SMI (Simulation Model Interface) so that it is registered in the simulator environment. Note that this is done once during the build process of the simulator (in the ModelEditor) and once during startup of the simulator.
The `initialize` method may look like this:

```
bool Simple::initialize(const char *name)
{
  ObjectID_t  object_id;
  ServiceID_t update_id;

  // Publish the instance using the 'this' pointer
  SMIPublishObject(name, this, &object_id);
```

```
    // Publish the data member 'foo'
    SMIPublishData(object_id, "foo", SMI_I32, &foo, 1,
                   SMI_INOUT_MODE, SMI_VIEW_TAG);

    // Publish the wrapper to the update method
    SMIPublishService(object_id, "update",
                      update_wrapper, &update_id);

    // Add the update method to the schedule
    SMIEnvAddServiceToSchedule(1000000000LL, 0, object_id,
                               update_id, NULL);

    return true;
  }
```

The call to `SMIPublishObject` publishes the object instance using the C++ `this` pointer. It gets registered under the name that was passed as argument to the `initialize` method. The result is stored in `object_id`. Note that no error checking is performed in this example, but you are strongly encouraged to do so in your own model source code.
Next, the only data member `foo` is published to the environment using the `SMIPublishData` function. The `object_id` is passed to tell the environment that `foo` is a member of our class instance.
The `update` method is published by means of a call to `SMIPublishService`. As explained earlier, we pass this function a pointer to the wrapper function instead of a pointer to the update method itself.
The `SMIEnvAddServiceToSchedule` adds the `update` method to the schedule. Note that you can leave this call out and use the EuroSim schedule editor instead to add the entrypoint to the schedule.
The next step is to create an instance of our 'Simple' class and call the `initialize` method. We do this in a separate function that can be called from ordinary 'C' source:

```
  extern "C" bool publish_simple(const char *name)
  {
    Simple *simple = new Simple();
    return simple->initialize(name);
  }
```

Next we create an interface header file `simple.h`:

```
  #ifdef __cplusplus
  extern "C" {
  #endif
    void publish_simple(const char *name);
  #ifdef __cplusplus
  }
  #endif
```

If you do not already have a file that contains the `ManagerPublishInterfaces` function, then create a file called `modelmanager.c` and add the following lines to it:

```
  #include <smi.h>
  #include "simple.h"

  void ManagerPublishInterfaces(void)
  {
    /* initialise the SMI. */
    SMIInitialise();

    /* Create an instance of class 'Simple' and publish it */
    publish_simple("Simple");
  }
```

Unless you need additional initialization in your model(s), we can suffice with an empty implementation of `ManagerInitialise`:

```
  void ManagerInitialise(void)
  {
  }
```

Add `modelmanager.c` and `simple.cpp` to your model in the ModelEditor and then select the *Build All* command in the *Tools* menu. Make sure that you selected SMP support in the *Build Options*.
The next step is to run the ScheduleEditor to create a custom schedule or to run the SimulationController and use the generated schedule. The published variables and entrypoints can be found under the `SMP` org node in the data dictionary.

Part III

EuroSim Advanced Topics

Chapter 16

Hardware Interfaces to EuroSim [1]

16.1 Introduction

With EuroSim Mk4.0 HIL it is possible to use three different hardware interfaces, the External Interrupt Interface, the MIL1553 Interface, and the Serial Interface. The External Interrupts interface provides access to the external interrupt functionality on SGI Origin/Challenge/Onyx L/XL. The MIL1553 interface, provides a general interface to access a MIL-STD-1553 serial data bus. The Serial interface provides non-blocking access to the standard RS232/RS422 interface on SGI Origin/Challenge/Onyx L/XL Each of these interfaces will be described in the following subsections.
Example code from a complete demonstration model, using MIL1553, serial and external interrupts library calls, is available and installed in `$EFOROOT/src/Mil1553Model`.

16.2 External interrupts interface

The External Interrupt interface provides the following services:

- EuroSim tasks can generate output interrupts to one of the four SGI output connectors;
- A standard EuroSim input connector "EI" will be raised by the scheduler for each incoming input interrupt. Note that the two input connectors are hardware short-circuited, thus in fact only one interrupt exists.
- A user can install a user defined handler for incoming input interrupts.
- Input interrupts can be used as the (external) real-time clock of the scheduler, instead of the default itimer mechanism.

16.2.1 Generation of output interrupts

The External Interrupt interface provides the function *esimEIPulse* to generate a pulse on one of the four output connectors of the SGI Origin/Challenge/Onyx. Because the generation of a pulse will never block, this function can be directly called from any task, in any state. For more information, refer to the *esimEI* manual page.

16.2.2 The input connector "EI"

The scheduler is invoked by a "heartbeat" mechanism, which is default an internal timer mechanism running at 100 Hz. This function will check if there are queued input interrupts.Each time that an interrupt is received, the "EI" input connector will be raised, which will cause execution of the attached user task. If several interrupts are received in quick succession ($>$100 hz), then the raising of the associated "EI" input connectors will be delayed until the following timeslots (see Figure 16.1).

[1] Not supported on the Windows NT platform.

For more information on the “EI” input connector, see also Section 11.3.4.2.

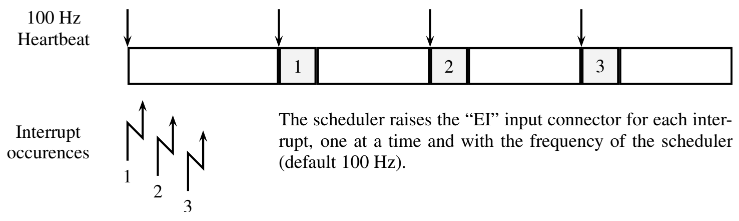

100 Hz
Heartbeat
1
2
3
Interrupt
occurences
1
2
3
The scheduler raises the “EI” input connector for each interrupt, one at a time and with the frequency of the scheduler (default 100 Hz).

Figure 16.1: Raising EI Input Connector

16.2.3 User defined interrupt handler

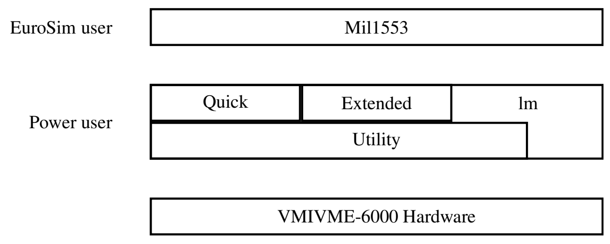
If “direct” response is required, the previous solution will not always be sufficient. The External Interrupt interface provides the functionality to directly activate a user defined handler on interrupt occurrences. On interrupt occurrences, the “`/dev/ei`” (or “`/dev/external_int/1`”) device will send a user defined signal to the scheduler, so that the interrupt handler will be executed directly.
The handler should be defined as a normal EuroSim task, and it can be installed via the schedule Configuration option (see Section 11.3.5, on the EI handler field of the Configuration option).
When interrupts will occur very fast after each other, some signals can be lost (UNIX anomaly) so that not all interrupts will be handled.
This mechanism interrupts the scheduler, and thus the real-time performance and/or (locking) behavior of the system. Be careful with this mechanism!

16.2.4 External real-time clock

The previous mechanism can be used as the heartbeat mechanism for the scheduler. This is done by using the interrupts to drive the scheduler clock. With this option the external interrupts will drive the scheduler instead of the internal timer.
This option is selected via the schedule Configuration option (see Section 11.3.5, on the ClockType field of the Configuration option).
When the scheduler is driven by the external interrupts, the user cannot install a user defined interrupt handler, because this will overwrite the existing scheduler heartbeat handler which causes a scheduler hangup.
The default basic frequency of the scheduler is 100 Hz, which means that one interrupt stands for 10 ms in real-time.

16.3 MIL1553 interface

The objective to the EuroSim MIL1553 interface is to offer the user a set of functions that will allow him to communicate with the devices on the bus, without having to bother about the details involved with programming the MIL1553 communication itself. A power user should however be able to employ all functionality and features provided by the protocol or by the hardware interface card, e.g. program the card to simulate a MIL1553 bus with one or more RTs (VMIVME-6000). The MIL1553 interface provides both interface levels.



The VMIVME-6000 BCU software library (quick, extended and utility) offers all the functionality and features that are provided by the VMIVME-6000 board. The library is ported to IRIX 6.5 and is (partly) tested on SGI/Onyx. For more information about this library see [VMI] and [VMI93]. For optimisation and engineering reasons, a new software module (lm) was developed, and added to the BCU software library.
The MIL1553 interface is built on the lm interface, and will be used by the normal EuroSim user. It provides the general interface to a MIL1553 device. The services of the MIL1553 interface are described in the following subsections.

16.3.1 Scenarios

The MIL1553 interface is based on a "scenario driven" mechanism (called "control blocks" for VMIVME-6000). The user firstly defines (e.g. in the initialization phase) the input and output buffers (for RT mode), or defines the message transfers (for BC mode) that are required for the application model. Once activated, the MIL1553 interface will perform solitarily the specified message transfers (BC mode) and will send or receive the output/input buffers in reaction of BC write/read transfer actions (RT mode).

16.3.2 General operations

The MIL1553 object provides general services to open and close the MIL1553 device, and to let EuroSim operate in bus controller (BC), remote terminal (RT) or bus monitor (BM) mode. These general operations are `esimMil1553Open`, `esimMil1553SetMode`, `esimMil1553Close`, `esimMil1553Start`, `esimMil1553Poll` and `esimMil1553Stop`.
For detailed information see the *esimMil1553* manual pages or [PMA05].

16.3.3 Bus Controller operations

In BC mode the user can use the operation `esimMil1553BcAdd` to add "RT→BC", "BC→RT" and "RT→RT" message transfers (control blocks) to the scenario. This scenario can be started by the user with `esimMil1553Start`. BC scenario's are automatically stopped when the transfers are completed. With the `esimMil1553BcRead` and `esimMil1553BcWrite` operations, data can be read from / written to the scenario, without changing the scenario. For detailed information see the *esimMil1553* manual pages or [PMA05].

16.3.4 Remote Terminal operations

In RT mode the user can specify with the `esimMil1553RtAdd` operation, which RT's and which subaddresses are simulated. With the `esimMil1553Start` operation, the RT's are activated until the user explicitly stops the RT's with `esimMil1553Stop`. Active RT's will solitarily send or receive data from the bus controller. With `esimMil1553RtRead` and `esimMil1553RtWrite`, the user can read and write the input/output buffers for each RT and subaddress, without stopping the RT's. For more information see the *esimMil1553* manual pages or [PMA05].

16.3.5 Bus Monitor operations

In BM mode the user can specify that the board should stop when the buffer is full, or can specify that the board will fill the buffer cyclic. Information of the monitored data can be obtained with the BCU utility library. For more information see the BCU utility manual.

16.3.6 Case study: Remote Terminal functions

```
/*
 * This example demonstrates the esimMil153 interface. In this example the
 * RT functionality will be demonstrated. The RT 3 that is simulated writes
 * incoming data on subaddress 5 to subaddress 10.
 */
#include <esim.h>
#include <esimMil1553.h>

#define MIL1553_BOARDNR 0
#define VME_A16_DEVICE "/hw/vme/1/usrvme/a16n/d16"
#define VME_A24_DEVICE "/hw/vme/1/usrvme/a24n/d16"

int main (void)
{
  int mil1553;
  uword buf[8];

  /*
   * Open the milbus device
   */
  if ((mil1553 = esimMil1553Open(MIL1553_BOARDNR, VME_A16_DEVICE,
                        VME_A24_DEVICE)) == -1) {
    esimError("Cannot open Mil0\n");
    return 1;
  }

  /*
   * Set the mode of the milbus to RT
   */
  if (esimMil1553SetMode(mil1553, MODE_MRTMON) == -1) {
    esimError ("Cannot set RT mode\n");
    return 1;
  }

  /*
   * Add RT 3 to control block
   */
  esimMil1553RtAdd(mil1553, 3);

  /*
   * Start the RT
   */
  esimMil1553Start(mil1553);

  /*
   * Read incoming data on subaddress 5 and write it to subaddress 10
   */
  esimMil1553RtRead(mil1553, 3, 5, buf, 8);
  esimMil1553RtWrite(mil1553, 3, 10, buf, 8);

  /*
   * Stop the RT
   */
  esimMil1553Stop(mil1553);

  /*
   * Close milbus and exit
   */
  esimMil1553Close(mil1553);

  return 0;
}
```

Note that in example given above (and in the remainder of this chapter), use is made of the `esimError` function. If the code is running on an external simulator, the equivalent `extError` is available (see the `extMessage(3)` man page).

16.3.7 Case study: Transferring data between a BC and a RT

```
/*
 * This example demonstrates the esimMil153 interface. In this example a
 * BC->RT transfer and RT->BC transfer is demonstrated. The BC->RT transfer
 * will send 8 words from the BC to RT 3 and subaddress 5.
 * The RT->BC transfer will receive 8 words from RT 3 and subaddress 10.
 */

#include <esim.h>
#include <esimMil1553.h>

#define MIL1553_BOARDNR 0
#define VME_A16_DEVICE "/hw/vme/1/usrvme/a16n/d16"
#define VME_A24_DEVICE "/hw/vme/1/usrvme/a24n/d16"

int main (void)
{
  int i;
  int mil1553;
  uword snd[8];
  uword rcv[8];


  /*
   * Open the milbus device
   */
  if ((mil1553 = esimMil1553Open(MIL1553_BOARDNR, VME_A16_DEVICE,
                                 VME_A24_DEVICE)) == -1) {
    esimError ("Cannot open Mil0\n");
    return 1;
  }
  /*
   * Set the mode of the milbus to BC
   */
  if (esimMil1553SetMode(mil1553, MODE_BCSIM) == -1) {
    esimError ("Cannot set BC mode\n");
    return 1;
  }

  /*
   * Add BC->RT transfer of 8 bytes from BC to RT 3 and subaddress 5
   */
  esimMil1553BcAdd(mil1553, 3, 5, BC_RT_TRANSFER, 8);
  /*
   * Add RT->BC transfer of 8 bytes from RT 3 and subaddress 10 to BC
   */
  esimMil1553BcAdd(mil1553, 3, 10, RT_BC_TRANSFER, 8);

  /*
   * Initialise and write the data to be send for BC->RT transfer
   */
  for (i = 0; i < 8; i++) {
    snd[i] = i;
  }
  esimMil1553BcWrite(mil1553, 3, 5, snd, 8);

  /*
   * Start the transfers
   */
  esimMil1553Start(mil1553);

  /*
   * Check whether all transfers (=the scenario) have finished.
   * This example uses a busy wait. Don't do that in a RT simulator.
   */
  while (esimMil1553Poll(mil1553) == 0);

  /*
   * Read and print the data of the RT->BC transfer
   */
  esimMil1553BcRead(mil1553, 3, 10, rcv, 8);
  for (i = 0; i < 8; i++) {
    printf("%d\n", rcv[i]);
  }

  /*
   * Close milbus and exit
   */
  esimMil1553Close(mil1553);
  return 0;
}
```

16.4 Serial interface

16.4.1 Design and operation

The Serial interface provides non-blocking read and write operations for standard serial devices. The Serial interface uses the standard IRIX 6.5 serial device drivers that already supports non-blocking. However, data must be buffered on read failures (when not enough data available). The Serial interface provides the initialization of the IRIX drivers and the buffering of data.
For detailed information, see the *esimSerial* manual pages or [PMA05].

16.4.2 Case study: Setting up a serial interface

```
/*
 * This example demonstrates the non-blocking read and write of the
 * esimSerial interface. Two serial devices are opened. From one device
 * 10 bytes are read and then sent to the other.
 */
#include <stdio.h>
#include <unistd.h>
```

```
#include <esimSerial.h>

int main(void)
{
 int ser1;
 int ser2;
 unsigned char buf[10];
 int buffered = 0;

 /*
  * Open the input device
  */
 if ((ser1 = esimSerialOpen("/dev/ttyd2", buffered)) == -1) {
  printf("Cannot open /dev/ttyd2\n");
  return 1;
 }

 /*
  * Open the output device
  */
 if ((ser2 = esimSerialOpen("/dev/ttyd3", buffered)) == -1) {
  printf("Cannot open /dev/ttyd3\n");
  return 1;
 }

 /*
  * Read non-blocking data from the input device
  */
 while (!esimSerialRead(ser1, buf, 10)) {
  printf("Non blocking read demo\n");
  sleep(1);
 }

 /*
  * Write the received data
  */
 esimSerialWrite(ser2, buf, sizeof(buf));

 /*
  * Close both devices
  */
 esimSerialClose(ser2);
 esimSerialClose(ser1);

 return 0;
}
```

16.5 External Events

16.5.1 External Event Sources

External events can be generated by:

- VME interrupts; interrupt level and vector must be specified.
- PCI interrupts.
- External Interrupts.

- POSIX named semaphores; see the manual page of sem_open. The semaphores can posted by any application on the same machine.
- Signals; available are the signal numbers between `SIGRTMIN` and `SIGRTMAX` that are not used by EuroSim internally.
- EuroSim Compatible Devices; these are devices which have device drivers adapted for EuroSim. These devices provide specific ioctl functions which EuroSim uses to wait for interrupts. See Section 16.5.3.

16.5.2 Event dispatching

There are two type of external event handlers: automatic and user defined (see Figure 11.3.5).

Automatic handlers
: can be used if the external event handler is connected to exactly one EuroSim input event (which can have one input connector in every state). Every time an external event arrives the input connector in the active state is raised. No additional code is required.

User defined handlers
: allow more input connectors to be associated to one external event source. It also allows a faster response to the external event in the interrupt handler or dispatcher code. The user should write this dispatcher code. This code cannot make use of EuroSim scheduling functions and only part of the EuroSim services can be used. (See esimEH manual page.)

User code can also be executed in interrupt handlers of VME, PCI or external interrupts. Such an interrupt handler can be needed to clear the HW interrupt. It also has a very short response time and user code can be executed while the HW interrupt line is still active. From the interrupt handler the external event can be forwarded to the dispatcher. Read the esimEH manual page for installation and restrictions on interrupt handlers.
Note: it is possible to install an external event handler that does not raise any EuroSim event, but only communicates through global data with model code.
Example of external event handler user code:

```
#include <string.h>
#include "esimEH.h"
#include "esim.h"

#define START_ID 0
#define STOP_ID 1
#define SHUTDOWN_ID 2
#define ERROR_ID 3

#define START 0x04
#define STOP 0x05
#define PANIC 0x10

#define STREQ(a,b) (!strcmp(a,b))

enum hw_status {
  DO_RESET,
  INTERRUPT
};

extern enum hw_status hw_status_get(void);
extern void hw_reset(void);
extern int hw_data_get(void);
extern void hw_shutdown(void);
```

```
static int associate(const char* name, void *user_data)
{
  (void) user_data; /* not used */
  if (STREQ(name,"START")) {
    return START_ID;
  } else if (STREQ(name,"STOP")) {
    return STOP_ID;
  } else if (STREQ(name,"SHUTDOWN")) {
    return SHUTDOWN_ID;
  } else if (STREQ(name,"HW_ERROR")) {
    return ERROR_ID;
  } else {
    return esimEH_NOT_ASSOCIATED;
  }
}

static int intr_handler(esimEH *context, void *user_data)
{
  enum hw_status status = hw_status_get();

  (void)user_data;

  if (status == DO_RESET) {
    hw_reset();
  } else {
    esimEHForward(context, &status, sizeof(status));
  }
  return 0;
}

static int dispatcher(esimEH *context, void *user_data, const void* msg,
                      int size)
{
  enum hw_status status = *(enum hw_status*)msg;
  int data;

  (void) user_data;      /* not used */
  (void) size;           /* not used */

  data = hw_data_get();
  switch (status) {
  case START:
    esimEHDispatch(context, START_ID, &data, sizeof(data));
    break;
  case STOP:
    esimEHDispatch(context, STOP_ID, &data, sizeof(data));
    break;
  case PANIC:
    hw_shutdown();
    esimEHDispatch(context, SHUTDOWN_ID, &data, sizeof(data));
    esimEHDispatch(context, ERROR_ID, &data, sizeof(data));
    break;
  default:
    break;
  }
  return 0;
}

static void dissociate(const char *name, int id, void *arg)
{
  (void)name;
  (void)id;
  (void)arg;
}

/* function for event handler installation */
int event_handler_install(void)
{
  return esimEHInstall("HW_INT",associate, dissociate, intr_handler,
                  dispatcher, NULL);
}

/* function for event handler uninstallation */
int event_handler_uninstall(void)
{
  return esimEHUninstall("HW_INT");
}

/* entrypoint raised by "START" */
void started(void)
{
  int hw_data;
  int size = sizeof(hw_data);
  esimEventData(&hw_data, &size);
  esimMessage("HW started: data = %d",hw_data);
}
```

16.5.3 User Defined EuroSim compatible devices

There are currently three devices with EuroSim compatible device drivers.

- Datum IRIG-B (bc635PCI)
- SBS PCI-VME bridge (Model 616/617)
- VMIC Reflective Memory (VMIPCI-5565)

Drivers for these devices are delivered as part of EuroSim for IRIX and Linux (except for the reflective memory device which is only supported under Linux).
It is possible to develop your own EuroSim compatible device drivers. The driver must implement three ioctl() commands: `OS_IOCTL_WAITINT` (95), `OS_IOCTL_BREAKWAITINT` (96) and `OS_IOCTL_GETIRQ` (97). These commands are defined in `osIntr.h`. The `OS_IOCTL_GETIRQ` command is Linux specific and optional.
The call to `ioctl(OS_IOCTL_WAITINT)` must wait for an interrupt or an event to arrive. It is done in a special event handler thread and must block forever if needed. It can only return on two occasions: an incoming interrupt (or event) or after an ioctl call with parameter `OS_IOCTL_BREAKWAITINT`. Whenever the call returns EuroSim expects that an interrupt (or event) has arrived.
The call to ioctl with command `OS_IOCTL_BREAKWAITINT` is issued when the application exits or when the user calls `esimEHUninstall()`. This ensures that the thread blocking on the `ioctl(OS_IOCTL_WAITINT)` can terminate properly.
When running the Linux OS the call to ioctl with command `OS_IOCTL_GETIRQ` is issued when the event handler is installed. If implemented, then this ioctl returns the IRQ number used for interrupts sent by this driver. This IRQ number is used by EuroSim to ensure that the interrupts go only to the CPU where the event handler is running.

Chapter 17

Modelling a TM/TC Link

17.1 Introduction

With EuroSim the possibility exists to model a telemetry/telecommand link at run time either between two sub-models within a EuroSim simulator, or between EuroSim and another (external) computer.
The main feature of this library is to simulate the bandwidth and time-delay that characterize a long-range communication link, such as the TM/TC link between a ground station and a satellite. By default this delay is disabled and packets are forwarded without any delay. The TM/TC mechanism uses a central server process running within EuroSim, via which the two terminators (or clients) can communicate. The server can maintain one or more client-to-client links; links are bidirectional and can be established between any two internal clients or between an internal and an external client. For the latter, use is made of TCP/IP. No link can be established between two external clients.
EuroSim has the flexibility to handle any data structure as a packet: "packets" do not have to be compliant with ECSS PUS [Sec03] standards in order to be sent and received over the TM/TC link. In Figure 17.1 a schematic of a TM/TC link between an external simulator and a EuroSim simulator is provided.

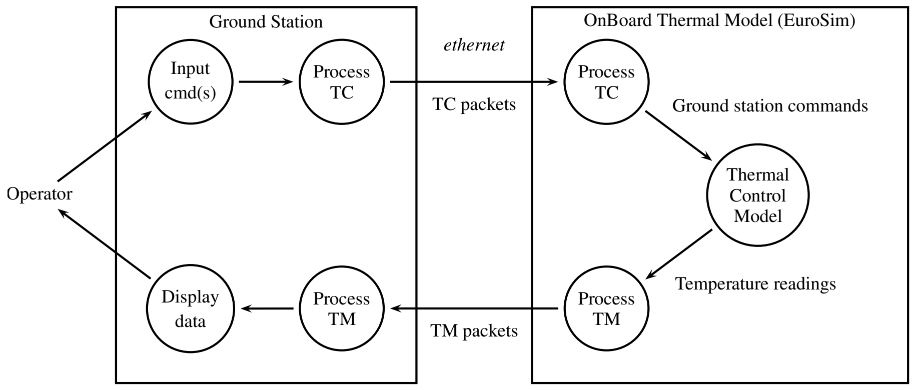


Figure 17.1: External TM/TC Link Schematic

In the case of an internal TM/TC link, i.e. between two or more sub-models within an EuroSim application, the link needs to be set up, customized and then used.
In the case of an external TM/TC client, the only difference is first to connect the external client over TCP/IP to the EuroSim server. Then the setting up and use of the link uses exactly the same routines. The EuroSim routines are intended to be usable within a heterogeneous environment, and should be suitable for any UNIX based simulators.

17.2 Characteristics of the TM/TC Link

Various characteristics of the link can be changed by calling `esimLinkIoctl` to customize the transmission of packets: some of the possible arguments for `esimLinkIoctl` are:

- LINKDELAY *integer*: sets the delay of the packages to the given number of milliseconds;
- LINKDELAYPROC *procname*: the given function will be called for each time a new package is put on the link with the call `link_write()`. Its return value is used as the delay for the given package. The arguments it gets are the Link identifier for which the delay is requested, and the last delay returned by this function for this Link;
- LINKBANDWIDTH *integer*: indicates the number of bytes which can be sent over the link per second. A negative number indicates 'Unlimited' (default) and will pose NO extra delay. When this value is set to 100, and 200 bytes are sent over, the package will take 2 seconds + the LINKDELAY to arrive at the other side. Note that the package will arrive as a single entity and therefore will not be visible (and cannot be read) as long as the complete package has not arrived;
- LINKMAXTIME_PENDING *integer*: indicates how many milliseconds data may be overdue in the queue before it is discarded. When a value less than zero is given, the packages will never be discarded (default).

The `esimLinkIoctl` procedure can be called at runtime, so can be used to introduce a variable delay time, for example in order to mimic an elliptical orbit or to simulate communications with a set of ground stations where the delay time is a function of the current location of the satellite.
Both TM/TC clients can set their own characteristics so that the upward and downward links can differ accordingly.
The *esimLink* manual page or [MAN05] provides more information on creating and customizing TM/TC links.

17.3 Summary of procedure

The following steps summarize how to set up and use a TM/TC link between two simulated stations:

1. If one of the simulators is external to EuroSim, set up a connection to the EuroSim server;
2. Create and customize the link between the two clients;
3. Send packets;
4. Receive packets;
5. Close the link.

17.4 Case study: setting up a TM/TC link

This section provides examples of how the procedure is implemented. The examples are taken from complete demonstration models (installed in `$EFOROOT/src/TmTc+ExtSimModel`).

17.4.1 Set up the external simulator as a EuroSim client

This only needs to be done if one of the clients for the TM/TC link is not a EuroSim application.
The external simulator is firstly linked to the EuroSim simulator as a client.

```
#include <extSim.h>

tmtcClient = extConnect(hostname, clientname, eventHandler, userdata,
                        async_flag, prefcon);
```

and takes the arguments:

- hostname: simulator host running target EuroSim simulator;
- clientname: name by which this client is to be known to the EuroSim server, e.g. "TMTC_Client";
- eventHandler: name of procedure in external simulator code which will process events coming from EuroSim simulator (flags indicating data updates and state changes are received as "events")
- userdata: pointer to user defined data. This pointer is passed to the eventHandler callback function.
- async_flag: flag to indicate that on incoming data the eventHandler callback function is to be called via a signal handler. If the flag is set to false, the user must call `extPoll()` whenever data arrives or periodically.
- prefcon: preferred connection on the EuroSim server; should be used to select between simulators when more than one is currently active on the server (default of 0 is sufficient if only one simulator is active)

17.4.2 Create and customize a link between the two TM/TC clients

The next step is to establish a (simulated) TM/TC link between the two "systems", i.e. either between the external client and EuroSim, or between two sub-models within a EuroSim application. In both cases, the link is set up and used in almost the same way.
The link needs to be created on both sides with `esimLinkOpen`. The function `esimLinkOpen` will initialize a link with the supplied name. A point-to-point link is not established until the other side has also called `esimLinkOpen` with the same link name. The pointer returned by `esimLinkOpen`(e.g. `tmtcLink` in the following example) is used as an identifier for the link in all future calls, e.g. read, write, close.
An external client needs to call `esimLinkConnect()` to connect the link to the simulator.
Various options can be set using `esimLinkIoctl`(see Section 17.2). In the first example here, the link for the ground station is set up and customized to "lose" packets if they arrive after a certain time delay:

```
#include <esimLink.h>

#define TMTC_CONNECTION_NAME "tmtc_connection"
#define REQ_FREQUENCY 10

static int gFrequency = REQ_FREQUENCY;

static void do_client(int signal)
{
  .....
 tmtcLink = esimLinkOpen(TMTC_CONNECTION_NAME, "rw");
 esimLinkIoctl(tmtcLink, LINKMAXTIME_PENDING, (1000/gFrequency)+100);
  .....
}
```

The following lines from the SpaceStation model give an example of setting up the link and using `esimLinkIoctl` to set the default delay for packets being transmitted:

```
#include "esimLink.h"
#include "esim.h"

#define TMTC_CONNECTION_NAME "tmtc_connection"

int requiredDelay = 300;   /* msec delay for link to ground */

void tmtcInit(void)
{
  ....
 tmtcLink = esimLinkOpen(TMTC_CONNECTION_NAME, "rw");
 if (tmtcLink == NULL) {
   esimMessage("Couldn't establish TmTc Link");
   return; /* Nothing possible */
 }
 esimLinkIoctl(tmtcLink, LINKDELAY, requiredDelay);
  ....
}
```

17.4.3 Sending packets

The packets are sent using `esimLinkWrite`, providing arguments for the link identifier, the data packet buffer, and the size of the buffer; e.g.:

```
send_packet(packet);

static int send_packet(EGSE_uns8 *pus)
{
   PUS_P_Header *header;

   header = (PUS_P_Header *)pus;
   return (esimLinkWrite(tmtcLink, (char *)pus,
            header->Packet_Length + PUS_P_HEADER_SIZE));
}
```

The packet is then available for the “other” client to read after a certain time delay, the length of the delay being dependent on the characteristics defined by `esimLinkIoctl`.

17.4.4 Receiving packets

Internal Client

In the example code for the EuroSim SpaceStation application model, a task is created which is scheduled at 5 Hz, and which calls a procedure which reads the (incoming telecommand) packets. The actually reading of the packets is done using `esimLinkRead`, the information being put into `tmtcPacket`. The return code `ret` is used to check on the success of the read:

```
#include "esim.h"
#include "esimLink.h"

#define PUS_P_HEADER_SIZE (sizeof(PUS_P_Header))
#define PUS_DATA_SIZE  512

static EGSE_uns8 *tmtcPacket = NULL;

void decodeTelecommand (void)
{
 int ret;
  ......
 ret = esimLinkRead(tmtcLink, (char *)tmtcPacket,
                PUS_P_HEADER_SIZE + PUS_DATA_SIZE);
 if (ret <= 0) {
   if (ret < 0) {
     /* incoming package is bigger than allocated data area */
     esimMessage("Fatal: TmTc command too big to read");
   }
   /* value of zero means no data to read */
   return;
 }
```

```
  PUS_P_Decompose_Packet_Header(tmtcPacket, &VersionNumber, &Type,
                             &DataFieldHeaderFlag,
                             &ApplicationProcessId,
                             &SegmentationFlags,
                             &SourceSequenceCount,
                             &PacketLength);
   ....
}
```

External Client - polling for packets

An external client has two possibilities to get packets. The first is to use the polling method as described above, i.e. regularly calling `esimLinkRead` to check if there are any incoming packets available. In the Ground Station TM/TC example code, incoming telemetry packets are checked for at 10Hz:

```
#include "esimLink.h"

#define REQ_FREQUENCY 10

static int gFrequency = REQ_FREQUENCY;
startTimer(gFrequency, do_client);

static void do_client(int signal)
{
  char buf[BUFSIZ];
  int n;
   .....
  n = esimLinkRead(tmtcLink, buf, BUFSIZ);
  if (n != 0) {
    browse_pus(buf, n);
  }
   .....
}
```

External Client - event driven response

Alternatively, use can be made of the events which are sent from the EuroSim server to the client to trigger a response directly as a result of an incoming packet. After `extClientConnect` or `evcConnect` has been called (see Section 17.4.1), the client automatically receives events signalling new link data (event-> type of `evLinkData`). These incoming events are passed to the procedure which was specified as part of the connect call. In this example, this facility is not used, but an action to do something with the incoming packet could easily be defined in the client's eventhandler (e.g. replacing the `DEBUGSTR` statement in the following extract):

```
static int eventHandler(Connection *conn,
                  const evEvent *command, void *data)
{
   switch (evEventType(command)) {
     case evShutdown:
       fprintf(stderr, "\nServer sent abort()\n");
       cleanup(0);
     case evLinkData:
       DEBUGSTR(("Incoming data"));
       break;
     default:
       DEBUGSTR(("Incoming unknown event `%d'",
              evEventType(command)));
   }
   return 0;
}
```

17.4.5 Close down link

If one of the clients is an external simulator, then the appropriate disconnect should be called (depends on which version of connect was used at the beginning (see Section 17.4.1):

```
#include <extClient.h>

extDisconnect(sim);
```

Then the TM/TC link can be closed:

```
#include "esimLink.h"

esimLinkShutdown();
```

Chapter 18

Interfacing external simulators to EuroSim

18.1 Introduction

With EuroSim the possibility exists to share simulation data at run time between EuroSim and another (external) simulator. This is achieved by linking external simulator (local) variables to (a subset of) the EuroSim data dictionary variables. During the simulation, EuroSim tools ensure that the values in the two data dictionaries are “mirrored” (the update frequency being a user definable parameter). In Figure 18.1 a schematic of the connection and associated functions for handling the data dictionary values is provided.

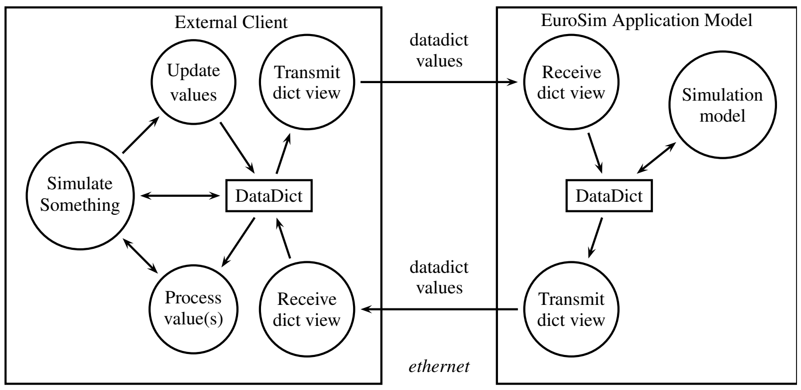


Figure 18.1: External Simulator Access Schematic

The two simulators need to be connected via a TCP/IP link. The EuroSim extSimAccess routines are intended to be usable within a heterogeneous environment, and should be suitable for any UNIX based simulators.
In addition to the specified data dictionary values, the external simulator also receives the simulation time, (elapsed) wall clock time and simulation state from EuroSim.

18.2 Selection of shared data items

It is possible to make all EuroSim data dictionary items accessible between the two simulators, but for performance, security or other reasons, it is often a requirement to limit the number of data items which are shared between the two. There are two levels of filter which can be applied:

- The EuroSim application decides which data dictionary items are to be visible to an external simulator, and whether with read and/or write access (defined in an `exports` file).

- The external simulator application decides which data dictionary items are to be used from those made available from EuroSim, and the type of access (defined in a data view).

An example is shown in Figure 18.2. The first box shows all the API items in the EuroSim data dictionary, i.e. all possible data items which can be shared between the two simulators. The middle box shows that by listing a subset of these items in a `exports` file, the EuroSim application can limit the number of data items which it wants the external simulator to share. In addition, read/write access can be limited. And finally, the third box for the external client shows how only some of the "public" data variables are actually referenced by the external simulator for his own internal use.

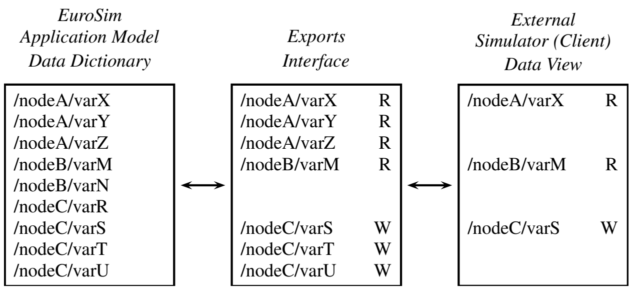


Figure 18.2: Filtering of Data Dictionary Items

18.3 Exports file

To share data between a EuroSim application and an external simulator, an additional file is needed. This file shall have the extension `.exports`, e.g `thermo.exports` and shall be included in the simulation definition file for this simulator.
The `exports` file specifies the EuroSim data dictionary items which will be made accessible at runtime to the external simulator (see Section 18.2). It is a text file and the contents can contain any number of lines of either of the following formats:

```
# this is an optional comment line;
# the next line can be tab or space separated
dict_node_ref  viewName  accessType
```

The `dict_node_ref` is a reference to a node or individual data item within the data dictionary hierarchy. Hence `/myNode/file.c/stateVariableA` is a legal reference which allows a particular variable to be accessed explicitly, as is `/myNode` which implicitly allows access to all of the data items under the named node in the data dictionary hierarchy.
The "`viewName`" provides a symbolic name for this set of data items, which needs to be referenced later on when creating a local data view for the external simulator. Each "`viewName`" has to be unique. It is generally recommended to make at least two views, one allowing read and one allowing write access. By choosing the views so that they contain different sets of data, this approach helps to reduce potential data inconsistencies which could be caused by simultaneous read/writes. Additional views may also be created for the purposes of data hiding, e.g. defining two views which give read access for nodes /A and /C, leaving node /B inaccessible to an external simulator.
The `accessType` indicates the type of access ("R" or "W") which the external simulator is given to the specified variables.

18.4 Creating multiple local data views

Instead of providing the external simulator with a single view of the shared EuroSim data (which is the situation implied in Figure 18.2 above), it can sometimes be advantageous to create and use several local data views. This is often useful when the external simulator is complex, and there are a significant number of shared data items. Two particular circumstances where multiple data views are recommended are:

- Data items need to be read/written at a wide range of frequencies, and therefore it is more efficient to split data items into views which can be given high and low update frequencies as required.
- The simulator model uses an object-oriented approach, and creating a data view per "object" continues to support this methodology.

Each local data view is created from (i.e. maps to) a single EuroSim data view (i.e. a single line as defined in the `exports` file). However, there can be several local data views mapping to a single EuroSim data view, for example to provide the possibility to read new values at different frequencies as mentioned above.

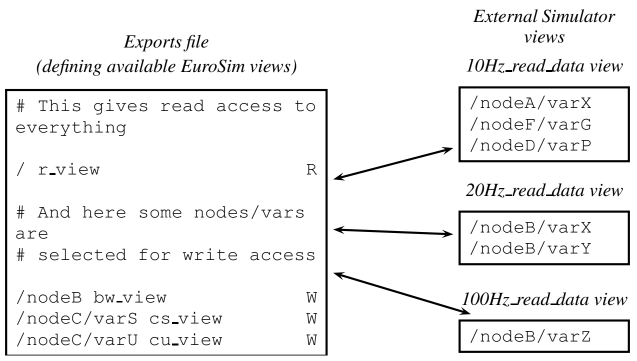


Figure 18.3: Mapping of EuroSim and Local Data Views

18.5 Synchronization

The external simulator access link can also be used to synchronize the client and the simulator with each other. If either the client or the simulator is slower than the other, the other side waits until the slowest side is finished. Also if one side stops for some reason, hitting a breakpoint or going to standby state for instance, the other side is halted as well.
The synchronization mechanism is coupled with the data being exchanged over the link so that data integrity is also ensured. At the simulator side this is done implicitly, but at the client side the user has to make sure to call `extViewSend()` before sending the synchronization token.
The synchronization token should be a unique token for each synchronization point in a simulator. It is possible to have multiple synchronization points in a simulator, possibly with multiple clients. It is the responsibility to make sure that the numbers are unique. If the same token number is used by multiple clients the simulator and/or one of the clients will very likely become blocked.
At startup the client shall connect to the simulator before the first synchronization token is sent from the simulator to the client. Sending synchronization tokens by the simulator is done by broadcasting as it is not possible to know in the simulator which client is performing synchronization (external simulator access clients are anonymous at the simulator side). So if the simulator sends the synchronization token before the client is connected, the token gets lost and the synchronization mechanism at the client side will have missed one token, resulting in a blocked client.
At the client side there are two functions for synchronization purposes. The function `extSyncSend()` sends the synchronization token to the simulator. The function `extSyncRecv()` waits for the synchronization token from the simulator.
The following example shows how to use the functions in a typical application where the client is two-way synchronized to the server.

```
#define SYNC_TOKEN 1234

extViewSend(write_view); /* send data to the simulator */
extSyncSend(sim, SYNC_TOKEN); /* send the synchronization token */
extSyncRecv(sim, SYNC_TOKEN); /* wait for the synchronization token */
```

At the simulator side there are two functions for synchronization purposes which are the counterparts of the functions on the client side. They differ from the client side functions by the fact that they do not have an argument to specify the connection. The function `esimExtSyncSend()` broadcasts the synchronization token to all clients. The function `esimExtSyncRecv()` waits for the synchronization token from the client.
The following example shows how to use the functions in a typical application where the simulator is two-way synchronized to a client.

```
#define SYNC_TOKEN 1234

esimExtSyncSend(SYNC_TOKEN); /* send the synchronization token */
esimExtSyncRecv(SYNC_TOKEN); /* wait for the synchronization token */
```

Please note that this method of synchronization cannot be used in a situation where hard-real-time performance is needed. The calls which wait for the synchronization token (`extSyncRecv()` and `esimExtSyncRecv()`) may block for a long time if the other side is stopped.
In Figure 18.4 a sequence diagram of the exchange of tokens is shown. Please note that the client as well as the simulator are always blocked from the point where they wait for the token until the next token is received. This is the essence of the synchronization mechanism.

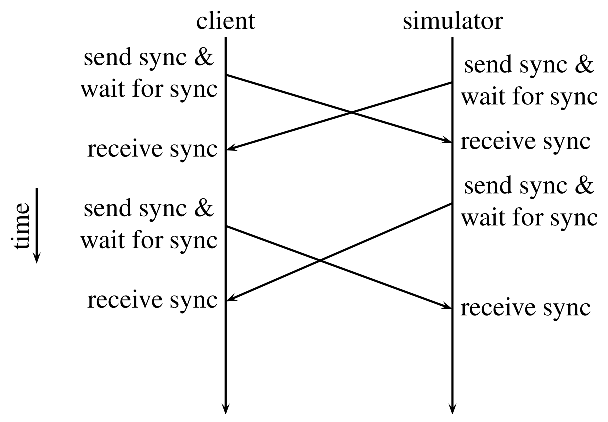


Figure 18.4: Synchronization sequence of a client and a simulator

18.6 Summary of procedure

The following steps summarize how to set up and use the connection between a EuroSim simulator and another (external) simulator:

1. Create an .exports file to specify which EuroSim data dictionary (API) items are visible to the external simulator.
2. Add calls to the external simulator code to link to EuroSim as a client at runtime.
3. Add calls to the external simulator code to make local data view(s) linking EuroSim data items to local variables.
4. Add calls to receive and send shared data at runtime.

5. Close the connection.

18.7 Case study: setting up shared data to another simulator

This section provides examples of how the procedure is implemented. The examples are taken from complete demonstration models (installed in `$EFOROOT/src/TmTc+ExtSimModel`).

18.7.1 Create an exports file

No changes need to be made to the EuroSim application model itself, but an additional file (`thermo.exports`) must be created and added to the simulation definition file using the simulation controller. The given example has the following lines in the `thermo.exports` file:

```
/        r_view          R
/SPARC   w_view          W
```

The first line in the export file specifies that all data items (i.e. from the root of the data dictionary downwards) are to be available to an external simulator with read only access and under the id of `r_view`. A specific node or data item can be referenced individually if required; however the "/" symbol is a useful shorthand to allow all data items to be referenced in one go.
The second line specifies that all data items under the SPARC node in the data dictionary are to be available under the id of `w_view` with write only access for the external simulator. As the SPARC node has also been included in the "/" specification for `r_view`, all data items under this note have effectively been given RW access and care should be taken when accessing their values.
In this example, two separate views are created for the external simulator: one containing data for reading, one containing data for writing. This is recommended to limit potential data inconsistency problems when allowing simultaneous read/write access.

18.7.2 Link the external simulator as a EuroSim client

The external simulator access library is initialized with the following call in the external simulator source code:

```
#include <extSim.h>

extInit();
```

Note that this only needs to be called once in your main() function.

Next, the link is set up with the following call in the external simulator source code:

```
#include <extSim.h>

sim = extConnect(hostname, clientname, eventHandler, userdata,
             async, prefcon);
```

and takes the arguments:

hostname
Simulator host running target EuroSim simulator.

clientname
Name by which this client is to be known to the EuroSim server, e.g. "TRPClientTester".

eventHandler
Name of procedure in external simulator code which will process events coming from EuroSim simulator (flags indicating data updates and state changes are received as "events").

userdata
: Pointer to user defined data. This pointer is passed as a parameter to the eventHandler procedure.

async
: Flag to indicate signal driven event handling versus polled event handling (see also extPoll).

prefcon
: Preferred connection on the EuroSim server; should be used to select between simulators when more than one is currently active on the EuroSim simulation server (default of 0 is sufficient if only one simulator is active).

A pointer is returned which identifies the simulator, and which you need to reference later on (e.g. when setting up the local view of the data dictionary). The `extClient` manual pages or [PMA05] provide more information on connection and disconnecting a client.

18.7.3 Determine host byte order

When accessing a simulator running on a host with a different byte order than the client the bytes need to be swapped. This is needed when simulator runs on an IRIX computer and the client runs on a Linux PC. In order to facilitate the detection of this difference in byte order, a message is sent to the client which allows you to determine the simulator byte order. Comparing the byte order to your own byte order will detect any differences. Below you find an example of such a detection routine:

```
int needs_swap = 0; /* set to 0 if the byte order is the same */

int eventHandler(Connection *conn, const evEvent *event,
                 void *userdata)
{
  switch (evEventType(event)) {
   case evExtByteOrder:
   {
     int size;
     int *magic;
     evEventArg(event, &offset, EV_ARG_RAW(&size, &magic));
     needs_swap = (*magic != EXT_BYTE_ORDER_MAGIC);
   }
  }
}
```

18.7.4 Set up local data view with links to EuroSim data

Overview

Once the client link is set up, local data view(s) can be created which link external and EuroSim data items. The views can contain all or a subset of the data items which were “exported” from EuroSim as described in Section 18.7.1.
There are three steps necessary, as shown in the following extract:

- Use of `extViewCreate` to create a local view.
- Use of `extViewAdd` to add a data item (local name + EuroSim name) to the view.
- Use of `extViewConnect` to connect the local view to the EuroSim simulator.

The `extView` manual pages or [PMA05] provide more information on data views.

```
#include "extSim.h"

#define VAR_VERBOSE_FLAG "Verbose"      /* Wonly */

void main(int argc, char *argv[])
```

```
{
  int ext_verbose = 0;
  ....
  w_view_ext = extViewCreate(sim, "w_view");
  ....
  extViewAdd(w_view_ext, VAR_VERBOSE_FLAG, &ext_verbose, extVarInt);
  .....
  extViewConnect(w_view_ext, EXT_DICT_WRITE, frequency,
                 COMPRESS_NONE);
  ....
}
```

Creating a Local Data View

In this example, only one local view (`w_view_ext`) is being created from the EuroSim `w_view` defined in the `model.exports` file. It is possible to make several views: for example `w_view_sensors`, `w_view_actuators`, each local view then having added to it a subset of the dict items available in the referenced `w_view` (see Section 18.4 for more information).

Linking EuroSim Variables, Local Variables in the Local Data View

In this case, the local view is to contain just one item: the EuroSim dictionary variable "Verbose". A define is used to make a symbolic name from the data dictionary variable name[1].
The link between the data dictionary (symbolic) name `VAR_VERBOSE_FLAG` and the local variable `ext_verbose` is made with the `extViewAdd` call. The type of the variable also needs to be made known when adding it to the view (e.g. `extVarInt`); the different types possible are listed in the `extView` manual pages, which also provides more detailed information on setting up data views.

Connecting the View to EuroSim

The final step is to define the connection characteristics for the local data view.
The given frequency in `extViewConnect` is only useful for views which are requested with `EXT_DICT_READ` permission. It indicates how many times per second a view must be sent over. The maximum frequency is the maximum frequency of the EuroSim scheduler (currently 200Hz). This frequency can be changed with a call to `extViewChangeFrequency`.
The last argument in `extViewConnect` indicates if compression should be used. For now only one compression method is available which simply discards values that are not changed since the last update and has minimal effect on process time.

Alternative Method to Create/Use a Local Data View

An alternative way of setting up a local data view requires access to EuroSim dict access routines, and the example code (as provided in `$EFOROOT/src/TmTc+ExtSimModel`) uses this technique to set up the write view. The method described above is generally preferred, as it can be used by any external simulator independently of EuroSim, the only knowledge of EuroSim then being required is a list of data dictionary items.

18.7.5 Receiving and sending shared data at runtime

Receiving Data Updates from the EuroSim Simulator

When the view is connected, events from the EuroSim server arrive automatically and are passed through the *eventHandler* which was specified when `extConnect` was called (see Section 18.7.2). Incoming events can either indicate that the data view has just been updated (`event->type` of `evExtSetData`) or

[1]At the moment the only way to find the correct data dictionary variable names is to get the information on-line from the *info* menu option in the Model Editor or DictBrowser, or to look in the model *.dict file.

that a state change command has been issued by EuroSim (`evEventType(event)` of `rtExecuting`, `rtStandby`, etc.). Each event is timestamped with simtime and runtime (`evEventSimTime(event)` and `evEventRunTime(event)` respectively).
In the given example external simulator code, an incoming evExtSetData event is used to trigger a display of the data (the data being read from local memory by the `curses_print` procedure). Similarly, any state events trigger the procedure `curses_state` which prints the state to screen together with the time-stamps. With this mechanism, if the update frequency is set to 10Hz, then the `curses_print` procedure is effectively being scheduled at 10Hz:

```
static int eventHandler(Connection *conn, const evEvent *command,
                void *userdata)
{
  char buf[50];
  evArgRec *pt;

  switch (evEventType(command)) {
    case evExtSetData:
      curses_print(command);
      break;

    case rtExecuting:
    case rtStandby:
      .....
      curses_state(command);
      .....
  }
  ...
}
```

However the events can be ignored: it is also possible to schedule (local) tasks which access the local data as and when required, rather than waiting for a data update to trigger processing.

Sending Updated Data Views

When the view is opened with write permission, the external simulator can send an update to the EuroSim simulator with a call to `extViewSend`. This will result in an event being sent to the EuroSim server (unless the data has not changed since the last call to `extViewSend`). The following example shows the verbose data item being toggled and the updated view being sent:

```
ext_verbose = !ext_verbose;
extViewSend(w_view_ext);
```

18.7.6 Close the connection

To close the connection between the client and the server, call the disconnect on exiting, for example:

```
installSignalHandler(SIGQUIT, cleanup);

static void cleanup(int signal)
{
  ...
  extDisconnect(sim);
  if (signal)
    exit(1);
  exit(0);
}
```

18.8 Performance

The External Simulator Access protocol is based on TCP/IP. This means that each data packet has some protocol overhead. When an isolated (peer to peer) Ethernet "network" is used (i.e. two computers connected by means of an Ethernet cross-over cable), then a theoretical throughput of 10 MByte/s can be achieved when using quality 100 Mbit/s network adapters and cable.
The above figure can be affected in a negative way by a number of causes:

- Cheap network cards that saturate the CPU at an interrupt rate that allows only a few MByte/s on a 100 Mbit/s network,
- Overhead and latency introduced by routers,
- Operating system,
- Driver implementation,
- Collisions in a non-isolated network,
- Configuration (tuning of TCP/IP parameters).

The latter point needs some explanation. On most systems, the default transmit and receive buffer size for TCP/IP sockets is only 16384 bytes. On Linux, you can use the `sysctl(8)` command to increase buffer sizes.

18.8.1 Maximum throughput

When using an non-isolated network, you must be aware of network "collisions" that affect the average throughput. Collisions are caused by multiple network nodes trying to access the medium (i.e. send a packet) at the same time. Each node will retry after a random interval. For that reason, a safe rule of thumb is to take one third (1/3) of the theoretical maximum as a basis for your calculations.
In practice this means that you can transfer 3 MBytes/s on a 100 Mbit/s network or, in EuroSim terms, have a view of 7500 long integers (4 bytes each) updated at 100 Hz. Be aware though that any network "hiccup" will cause buffer overflows at such high data rates.

18.9 Building the client

18.9.1 Unix and Linux

The external simulator access libraries are provided as dynamic shared objects (DSO's) and are part of the standard EuroSim distribution. The include files are located in the `include` subdirectory of the directory where EuroSim is installed.

18.9.2 Windows

When using the Cygwin environment, you can use the mingw gcc compiler and the external simulator access libraries as provided in the `lib` subdirectory of the directory where EuroSim is installed. If you prefer to use the Microsoft development tools, then you can use the DLL's. To link your client application with the DLL's, you must first create import libraries from the `.def` files in the `lib` subdirectory, for example:

```
lib /DEF:c:/eurosim/lib/libes.def
lib /DEF:c:/eurosim/lib/libesClient.def
```

The above commands create `libes.lib` and `libesClient.lib`, which you can use to link your client application with. Make sure that the DLL's can be found on the PATH before you execute your client application.

Chapter 19

Simulator Integration Support library reference

19.1 Introduction

The purpose of the Simulator Integration Support library is to support the integration of several independent models into one simulator without wanting to do the integration explicitly in (model) source code. In other words: the Simulator Integration Support library provides the “glue” between models.

19.2 Files

Two file types[1] have been introduced for this purpose:

- Model Description file
- Parameter Exchange file

Model Description files can be created and edited with the Model Description Editor, see Chapter 7. Parameter Exchange files can be created and edited with the Parameter Exchange Editor, see Chapter 8. The use of these files will be described in the following sub-sections by means of a use case example.

19.3 Use case example

19.3.1 Model files

Suppose we have two sub-models `modelA.c` and `modelB.c` as listed below.

Listing 19.1: The C source code for the `modelA` file node

```
#include <math.h>

static double x;
static double y;

void calc_sin(void)
{
  y = sin(x);
}
```

[1]The file extensions are provided in Appendix F.

Listing 19.2: The C source code for the `modelB` file node

```
static double counter;

void update_counter(void)
{
  counter = counter + 0.1;
}
```

The complete source code, including the other files discussed in this section, can be found in the `src` *subdirectory of the directory where EuroSim is installed.*

ModelA takes variable `x` as input to the `sin` function and stores the result in variable `y`. The entrypoint for the update of modelA is `calc_sin`.

ModelB takes variable `counter` as input, increments it and writes the result back to the same variable. The entrypoint for the update of modelB is `update_counter`.

When we want to use modelB to update the input variable of modelA, we would need to modify the source code of modelB to perform its update on variable `x` instead of using variable `counter`. We would also need to change modelA to remove the `static` keyword from variable `x` so that it can be accessed from modelB (global scope). When using the Simulator Integration Support library, we do not have to modify the source of the sub models as will be explained in the following sub-sections.

Figure 19.1 shows a screen shot of what the Model Editor looks like with the two sub-models *modelA* and *modelB*. The sub-models have been parsed and check marks are placed in front of the entrypoints and variables that have to be available in the data dictionary.

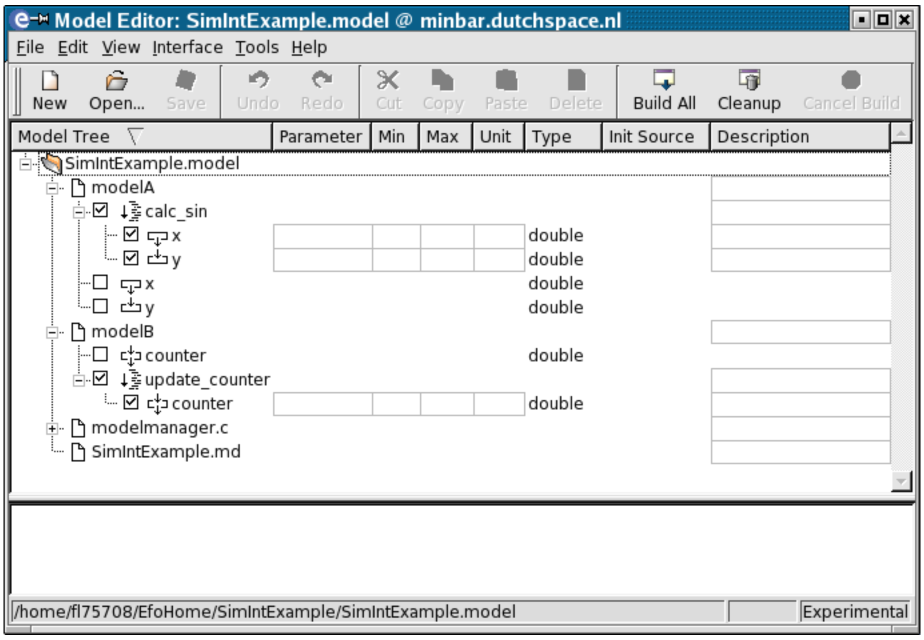

Model Editor: SimIntExample.model @ minbar.dutchspace.nl
File Edit View Interface Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete Build All Cleanup Cancel Build
Model Tree Parameter Min Max Unit Type Init Source Description
SimIntExample.model
modelA
calc_sin
x double
y double
x double
y double
modelB
counter double
update_counter
counter double
modelmanager.c
SimIntExample.md
/home/fl75708/EfoHome/SimIntExample/SimIntExample.model
Experimental

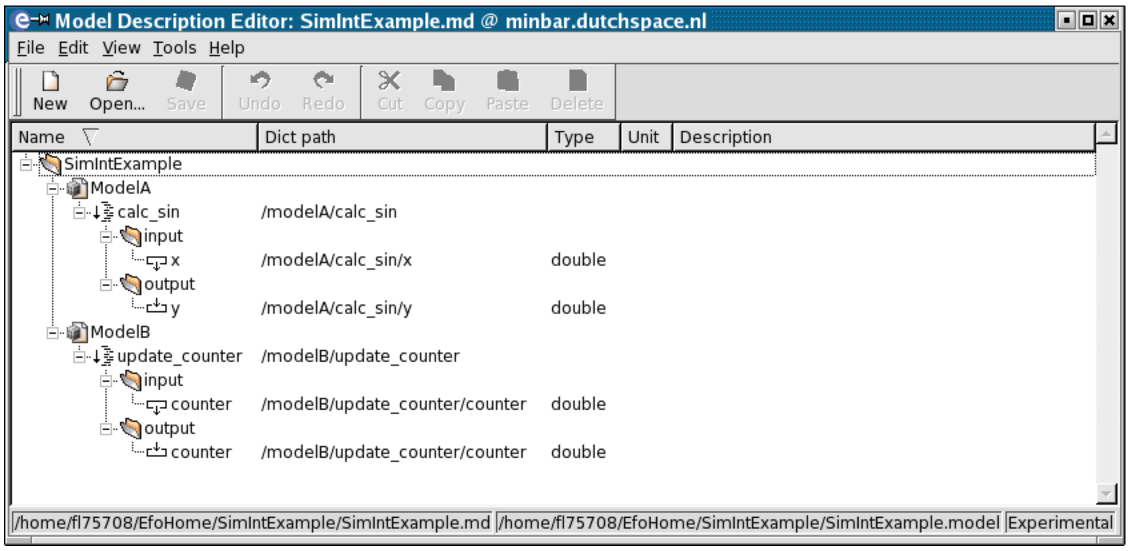
Figure 19.1: Model Editor

19.3.2 Model Description file

The philosophy behind the Model Description file is that each model has one or more input variables, one or more update functions (entrypoints) and one or more output variables. The Model Description Editor can be used to select the input and output variables and the entrypoints from the data dictionary and logically group them together, see for example the `calc_sin` node in Figure 19.2. This *describes* a model at a higher abstraction level even if the original model source code is rather unstructured or actually contains more than one sub-model. In the latter case, the Model Description file can be used to organize the model by defining multiple model nodes with entrypoints and variables that refer to a single model source code file. Each model variable that is described as a variable in the Model Description file will be available for exchange with other variable(s).

It is possible to add one or more Model Description file nodes to a model using the EuroSim Model Editor, see Section 6.2.3.4. When you select the *Edit* command on a Model Description file node in the Model Editor, the Model Description Editor will be started.

After specifying which variables from the example models should be available for model to model exchanges, the Model Description Editor looks like Figure 19.2. We have created two model nodes `ModelA` and `ModelB` that contain references to the entrypoints in the respective models. Since this is a very simple example, the screen shot shows an almost one to one copy of the original model tree in the Model Editor. Notice that the `counter` variable in the Model Description file has been duplicated to serve as an input variable as well as an output variable for ModelB.


Model Description Editor: SimIntExample.md @ minbar.dutchspace.nl
File Edit View Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete

Name	Dict path	Type	Unit	Description
SimIntExample				
ModelA				
calc_sin	/modelA/calc_sin			
input				
x	/modelA/calc_sin/x	double		
output				
y	/modelA/calc_sin/y	double		
ModelB				
update_counter	/modelB/update_counter			
input				
counter	/modelB/update_counter/counter	double		
output				
counter	/modelB/update_counter/counter	double		

/home/fl75708/EfoHome/SimIntExample/SimIntExample.md /home/fl75708/EfoHome/SimIntExample/SimIntExample.model Experimental

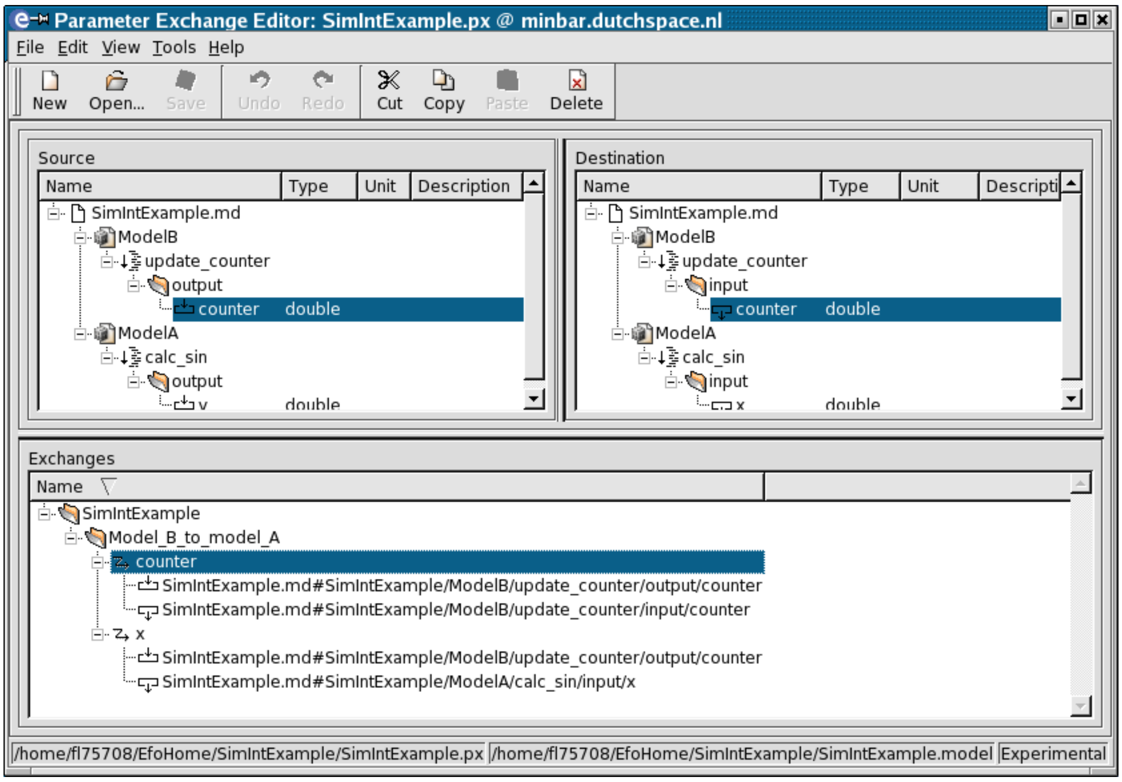
Figure 19.2: Model Description Editor

19.3.2.1 Datapool

Once you have finished editing a Model Description file, select the *Tools:Build All* menu command in the Model Editor, which generates the so called “datapool” (see also Section 7.2). The datapool contains the variables described in the Model Description file(s). It also contains automatically generated entrypoints to exchange the data between model variables and datapool variables. The variables in the datapool are always of the same type as the ones they refer to in the model files. During the build process, the variables and entrypoints in the datapool are merged into the data dictionary, see Section 19.6.

19.3.3 Parameter Exchange file

A Parameter Exchange file describes which output variables in the datapool should be copied to which input variables in the datapool. The input and output variables must be of the same type (and unit!). Parameter exchanges are grouped together in logical groups. For each parameter exchange group an entrypoint will be generated. Scheduling the parameter exchanges is described in Section 19.3.4. Use the Parameter Exchange Editor to create or modify a Parameter Exchange file. There is no need to re-run the build process in the Model Editor after creating or modifying a Parameter Exchange file, as the entrypoints are generated “on the fly” when the simulator is started.
For our use case example a screen shot of the Parameter Exchange Editor looks like Figure 19.3. Each time the parameter exchange entrypoint is scheduled, the value of output variable `counter` of ModelB is copied to input variable `x` of ModelA and to the input variable `counter` of ModelB. The parameter exchange entrypoint receives the same name as name the exchange group node. Thus, in our example the entrypoint will be available as “Model_B_to_model_A”.


Parameter Exchange Editor: SimIntExample.px @ minbar.dutchspace.nl
File Edit View Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete

Source

Name	Type	Unit	Description
SimIntExample.md			
ModelB			
update_counter			
output			
counter	double		
ModelA			
calc_sin			
output			
y	double		

Destination

Name	Type	Unit	Descripti
SimIntExample.md			
ModelB			
update_counter			
input			
counter	double		
ModelA			
calc_sin			
input			
x	double		

Exchanges

```
Name
SimIntExample
  Model_B_to_model_A
    counter
      SimIntExample.md#SimIntExample/ModelB/update_counter/output/counter
      SimIntExample.md#SimIntExample/ModelB/update_counter/input/counter
    x
      SimIntExample.md#SimIntExample/ModelB/update_counter/output/counter
      SimIntExample.md#SimIntExample/ModelA/calc_sin/input/x
```

/home/fl75708/EfoHome/SimIntExample/SimIntExample.px /home/fl75708/EfoHome/SimIntExample/SimIntExample.model Experimental

Figure 19.3: Parameter Exchange Editor

19.3.3.1 Why are Parameter Exchange files not part of the model?

This is done for flexibility. It allows the model developer to put together several sub-models into one simulator executable and describe the model variables by means of one or more Model Description files. The simulator developer could then create two Parameter Exchange files and reference these from two Schedule files. The first variant of the Parameter Exchange may for example update the input variables of one of the models with variables in the datapool that are updated by an external simulator (see Chapter 18). The second variant may update the input variables of one of the models with variables in the datapool that are updated by an internal model. In that way the test controller can easily switch between the two configurations, simply by selecting the appropriate Schedule file. The reason for having the Parameter Exchange file(s) referenced by the Schedule file is that the entrypoints are generated "on the fly" and you need the entrypoints when you edit the Schedule.

19.3.4 Specifying the schedule

As the last step when using Simulator Integration Support the schedule has to be specified. At this point we should have:

- A successfully built simulator executable,
- A successfully built data dictionary,
- One or more Model Description files (added to the model file as file nodes),
- One or more Parameter Exchange files (optionally added to the Project Manager).

We are now at a point were we can create the schedule file for the simulator. For our use case example a screen shot of the Schedule Editor looks like Figure 19.4.

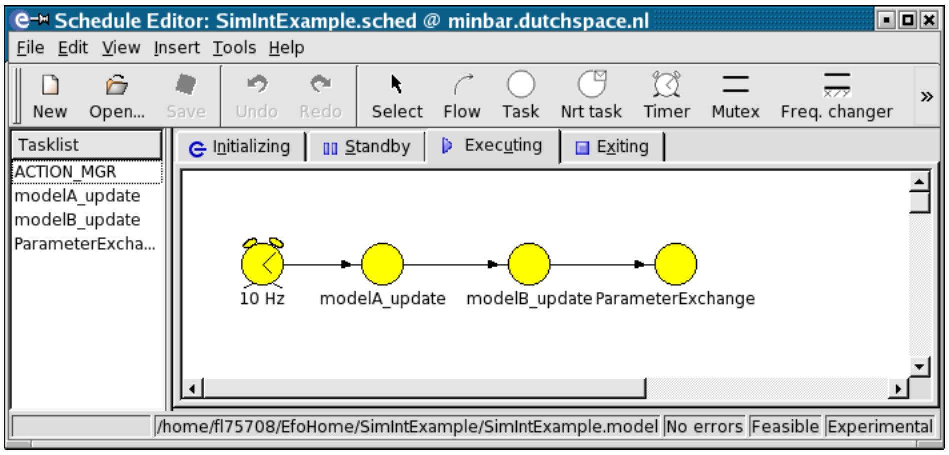

Schedule Editor: SimIntExample.sched @ minbar.dutchspace.nl
File Edit View Insert Tools Help
New Open... Save Undo Redo Select Flow Task Nrt task Timer Mutex Freq. changer
Tasklist
ACTION_MGR
modelA_update
modelB_update
ParameterExcha...
Initializing Standby Executing Exiting
10 Hz
modelA_update
modelB_update
ParameterExchange
/home/fl75708/EfoHome/SimIntExample/SimIntExample.model
No errors
Feasible
Experimental

Figure 19.4: Schedule Editor

Task *ModelA_update* contains three entrypoints:

- /datapool/SimIntExample/ModelA/calc_sin/input/set_input_variables
- /modelA/calc_sin
- /datapool/SimIntExample/ModelA/calc_sin/output/set_output_variables

The first entrypoint is generated by the Model Editor build process when the Model Description file was read. It copies variable `x` from the datapool to variable `x` of model A (step 1 in Figure 19.5). The second entrypoint is the one from model A and uses variable `x` in model A to calculate the sine value and store the result in variable `y` (step 2). The last entrypoint is also generated and copies variable `y` from model A to variable `y` in the datapool (step 3).

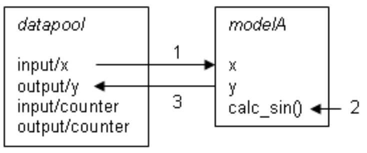


Figure 19.5: Datapool exchanges and update for model A

Task *ModelB_update* contains three entrypoints:

- /datapool/SimIntExample/ModelB/update_counter/input/set_input_variables
- /modelB/update_counter
- /datapool/SimIntExample/ModelB/update_counter/set_output_variables

The first entrypoint is generated by the Model Editor build process when the Model Description file was read. It copies variable `counter` from the datapool to variable `counter` of model B (step 4 in Figure 19.6). The second entrypoint is the one from model B and uses variable `counter` in model B to increment itself (step 5). The last entrypoint is also generated and copies variable `counter` from model B to variable `counter` in the datapool (step 6).

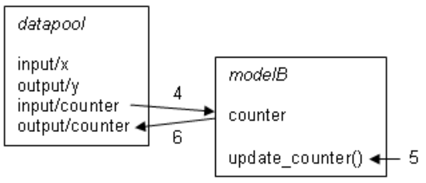


Figure 19.6: Datapool exchanges and update for model B

Task *ParameterExchange* contains one entrypoint:

- /paramexchg/Model_A_to_Model_B

This entrypoint copies the updated `counter` output variable in the datapool to the `counter` input variable and the `x` input variable (step 7 in Figure 19.7). After this parameter exchange the schedule starts again at step 1. This time model A uses the updated `x` variable to perform its model update.

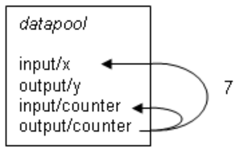


Figure 19.7: Parameter exchange

Notice that entrypoints that are generated for parameter exchanges are placed in a special node in the data dictionary called "paramexchg". The name of the entrypoint is the same as the name of the parameter exchange group node in the Parameter Exchange file. The parameter exchange entrypoint copies the values of the specified variable(s) from the source to the destination.

The names of the generated entrypoints to update the datapool and model variables receive the names of the input and output group nodes as specified by the Model Description file:

Name of entrypoint `:=` `set_`*nodename*`_variables`

In order to generate the parameter exchange entrypoints, you must use the *File:Parameter Exchange files* command in the schedule editor to specify which parameter exchange file(s) should be used by the simulator. As soon as you add a parameter exchange file, the Schedule Editor will automatically add the appropriate entrypoints to the internal data dictionary (it will not change the data dictionary file on disk), so that the entrypoints are available in the task and non-rt task dialogs. At run-time, i.e. when the simulator reads the schedule file, the referenced parameter exchange files are read and the entrypoints are also generated, but this time they will point to internal data structures that describe which datapool variables to copy.

19.3.5 Concluding remarks

During the use case example in the previous sub-sections we have seen that we can integrate two models without having to write or modify a single line of source code. Of course, in practice model source code may have to be modified in order to match variable types (in the example we used doubles for all variables).

19.4 Initial values

The variables in the datapool will receive the same initial value as specified in the data dictionary for the related model variable. Use initial condition files if you wish to set the datapool variables to different initial values.

19.5 Relation with SMP

The Simulator Integration Support library uses SMP functions (see Chapter 15) to publish the variables for the datapool, the datapool exchange entrypoints and the entrypoints for the parameter exchanges. When using SMP you must provide two initialization functions:

- `void ManagerPublishInterfaces(void)`
- `void ManagerInitialise(void)`

If you do not have SMP compliant models, then you can fulfill this requirement by implementing empty stub functions:

Listing 19.3: The C source code for the `modelmanager.c` file node

```
/* Empty stub functions for SMP initialization */

void ManagerPublishInterfaces(void)
{
}

void ManagerInitialise(void)
{
}
```

Add the `modelmanager.c` file as a file node to the model, see Figure 19.1.

19.6 Build process

Figure 19.8 shows the steps to build the simulator executable and data dictionary when using the Simulator Integration Support library. The build process (make) can be started from the Model Editor with the *Tools:Build All* menu command. First a data dictionary is generated from the model source code. This is the stage 1 data dictionary that is also used by the Model Description Editor. When the Model Description Editor is started from the Model Editor, the stage 1 data dictionary is always updated to ensure that all model variables are visible in the Model Description Editor. During the final build, i.e. when the Model Description file has been defined, the build process creates the datapool from the Model Description file(s) and merges its variables and entrypoints with the stage 1 data dictionary in order to create the final data dictionary. The final data dictionary will be used by the simulator and other EuroSim tools (such as the Schedule Editor).

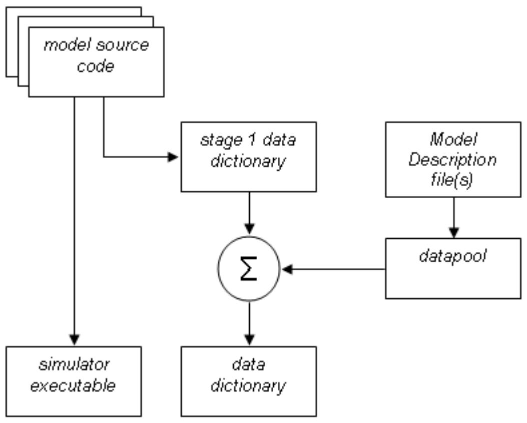


Figure 19.8: Build process steps

Chapter 20

COM Interface reference

20.1 Introduction

The EuroSim COM interface allows Windows scripting clients, such as Visual Basic for Applications (VBA) supported by MS-Excel, to launch and/or connect to an EuroSim simulator and control this simulator from your client application.

20.2 Installation

20.2.1 VBA

When you plan to use this interface only with VBA, then you only need to install the esimcom.dll (the actual component) and esimcom.tlb (the type library). If you have an installation of EuroSim for Windows, then these files have already been copied for you. If you want to use your own client applications on a PC that does not have EuroSim installed, then you can manually install the COM interface:

- Copy the esimcom.dll and esimcom.tlb files from the EuroSim bin subdirectory to an appropriate directory (this can be the system directory, usually `C:\windows\system32`).
- Copy the files pthreadGC.dll and oncrpc.dll to the system directory, if they are not installed on the target system yet (you can find these files in the system directory of a system that has EuroSim for Windows installed.
- Open a command shell and go to the directory where you just copied the two files.
- Type the following command at the prompt:

  ```
  regsvr32 esimcom.dll
  ```

  followed by the `Enter` key.

20.2.2 C++

When you plan to develop C++ clients, then you need the esimcom.h (the header file) and the esimcom_if.c files in your development environment. Copy those files from the EuroSim include directory into an appropriate directory and make sure you add an include path to the esimcom.h file. The esimcim_if.c file should be compiled and linked in with the other files of your client program.
If you get errors at compile time about MIDL versions, you may need to use the esimcom_vc6.h header file instead of esimcom.h.

20.3 Programmers reference

The complete reference is installed as HTML pages in the doc subdirectory of the directory where EuroSim is installed. For background information on COM/DCOM programming, see [COM98].

20.4 Use case – Excel example

In this chapter we will guide you through the EuroSim COM interface by means of a use case. We have a simulation, which is kept very simple for demonstration purpose and a client application, based on Microsoft Excel and Visual Basic for Applications, which will launch the simulator, monitor the state of variables in the model code of the simulator, monitor wallclock time, simulation time and the state of the simulator. As a last example, the client will change the value of the variables in the model code.

20.4.1 The simulator

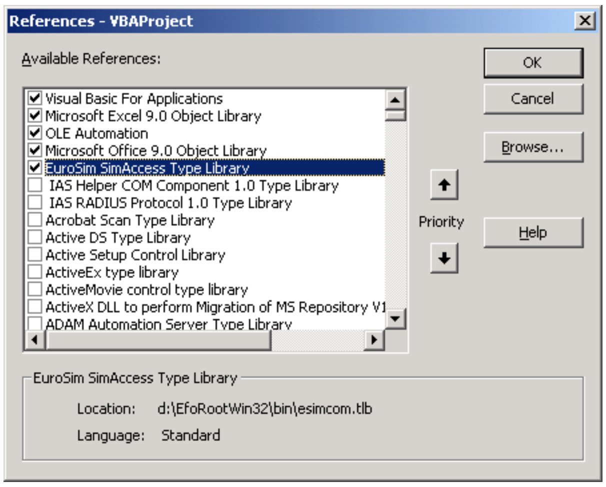
First we build the simulator from the src/com/counter directory. Start up the EuroSim project manager (either double click the EuroSim icon on the desktop or run **`esim`** from the command line). You may want to refer to the approriate section of the SUM if you are not yet familiar with the following steps:

- Create a project called ‘Counter’ and give it a directory.
- Copy the files from the src/com/testsim directory into your project directory.
- Add the Counter.sim and Counter.model files to the project.
- Double click the Counter.model file in the project manager and build (F8 key) the simulator with the Model Editor.
- Double click the Counter.sim file in the project manager and run the simulator by clicking the ‘Init’ button of the Simulation Controller. If all went well, you should see some messages in the log window of the Simulation Controller indicating that the simulation is running.
- Stop the simulation by clicking the ‘Stop’ button of the Simulation Controller.

At this point we have a working simulator, which we can use to test the MS Excel based client application.

20.4.2 The MS Excel client application

We create the MS Excel client application in a couple of steps. Open a new MS-Excel sheet and open the Visual Basic (VB) editor: Menu: *Tools:Macro:Visual Basic Editor*. Open the references dialog box in the VB editor: Menu: *Tools:References...* and check the box in front of the ‘EuroSim SimAccess Type Library’, see Figure 20.1. Press the ‘OK’ button of this dialog to accept the changes and close the dialog.



- [✓] Visual Basic For Applications
- [✓] Microsoft Excel 9.0 Object Library
- [✓] OLE Automation
- [✓] Microsoft Office 9.0 Object Library
- [✓] EuroSim SimAccess Type Library
- [ ] IAS Helper COM Component 1.0 Type Library
- [ ] IAS RADIUS Protocol 1.0 Type Library
- [ ] Acrobat Scan Type Library
- [ ] Active DS Type Library
- [ ] Active Setup Control Library
- [ ] ActiveEx type library
- [ ] ActiveMovie control type library
- [ ] ActiveX DLL to perform Migration of MS Repository V1
- [ ] ADAM Automation Server Type Library

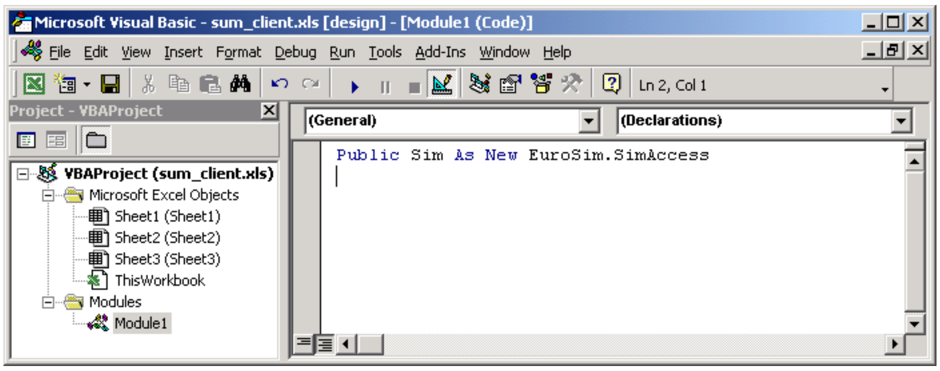
EuroSim SimAccess Type Library
Location: d:\EfoRootWin32\bin\esimcom.tlb
Language: Standard

Figure 20.1: Adding a reference to the EuroSim type library

Now we will add a declaration that creates an instance of the EuroSim COM interface. Create a new module by right-clicking the VBAProject tree and selecting *Insert:Module*. Then type the following in the code section of the new module:

```
Public Sim As New EuroSim.SimAccess
```

Your VB editor should now look similar to Figure 20.2.


Microsoft Visual Basic - sum_client.xls [design] - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Ln 2, Col 1
Project - VBAProject
VBAProject (sum_client.xls)
Microsoft Excel Objects
Sheet1 (Sheet1)
Sheet2 (Sheet2)
Sheet3 (Sheet3)
ThisWorkbook
Modules
Module1
(General)
(Declarations)

```
Public Sim As New EuroSim.SimAccess
```

Figure 20.2: Declare an instance of the interface

Go back to the Excel sheet and make sure you have a ‘Control Toolbox’ toolbar, see Figure 20.3 (if not, goto menu: *View:Toolbars* and check the ‘Control Toolbox’). Place a command button on the sheet and change its name to ‘Init’ (right click the command button and select: *CommandButton Object:Edit*). Hit the `Esc` key on the keyboard and double-click the command button. Enter the following code in the CommandButton1_Click subroutine:

```
Private Sub CommandButton1_Click()
   On Error Resume Next
   Sim.Launch "localhost", "C:\mysims\Counter", _
              "Counter.sim", "TestClient", 0
   If Err <> 0 Then
      MsgBox ("Error: " & Err.Description)
   Else
      [A5].Value = "Launch successful"
   End If
End Sub
```

Instead of ”C:\mysims\Counter” for the working directory, you should fill in the path that you used when creating the simulator project, i.e. the directory where the Counter.sim and Counter.model files are located.
In a similar way add the ‘Go’ button with the following code:

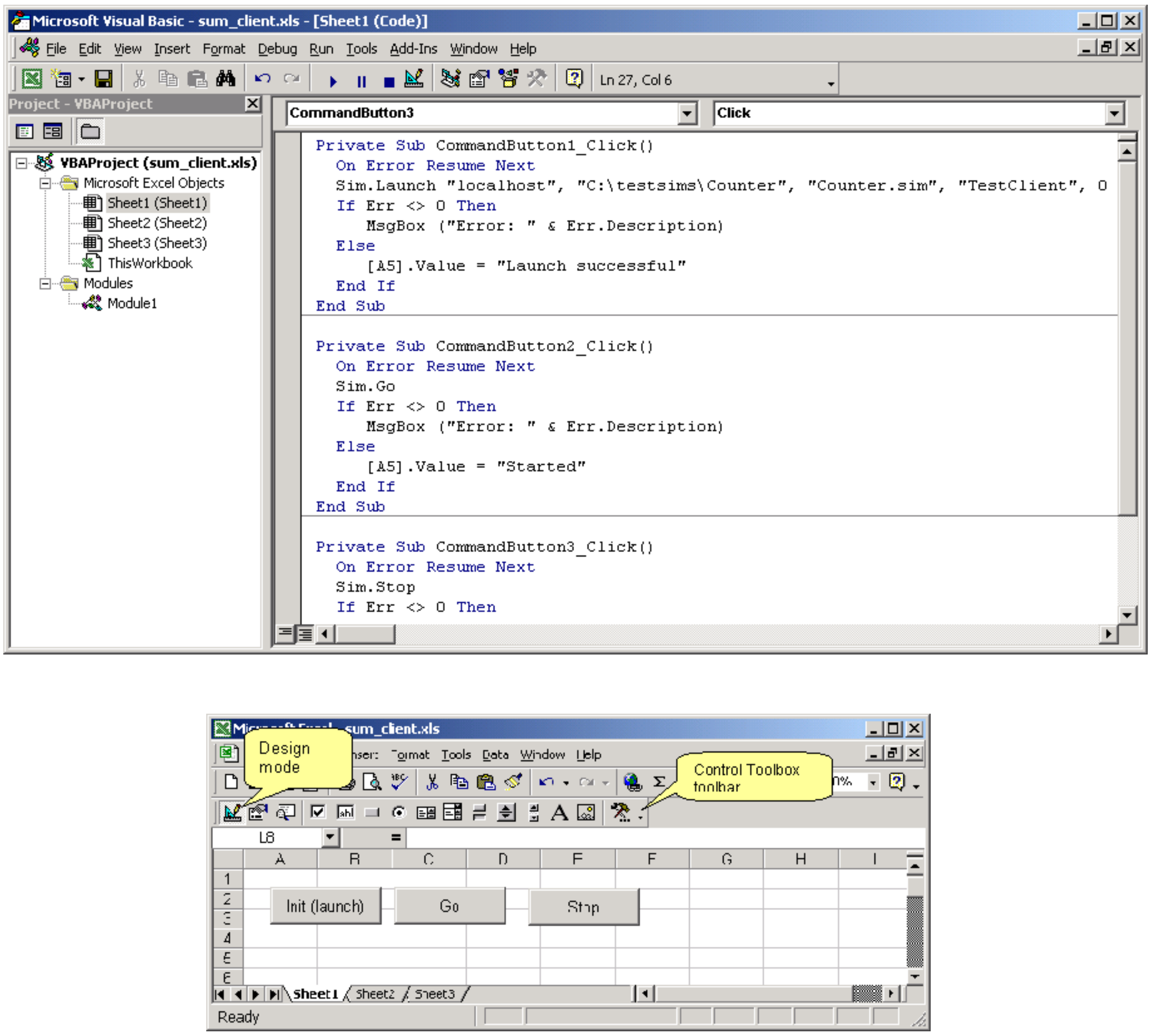
```
Private Sub CommandButton2_Click()
   On Error Resume Next
   Sim.Go
   If Err <> 0 Then
      MsgBox ("Error: " & Err.Description)
   Else
      [A5].Value = "Started"
   End If
End Sub
```

And a ‘Stop’ button with the following code:

```
Private Sub CommandButton3_Click()
   On Error Resume Next
   Sim.Abort
   If Err <> 0 Then
```

```
            MsgBox ("Error: " & Err.Description)
        Else
            [A5].Value = "Stopped"
        End If
    End Sub
```

Your client should now look similar to Figure 20.3.



```
Private Sub CommandButton1_Click()
   On Error Resume Next
   Sim.Launch "localhost", "C:\testsims\Counter", "Counter.sim", "TestClient", 0
   If Err <> 0 Then
      MsgBox ("Error: " & Err.Description)
   Else
      [A5].Value = "Launch successful"
   End If
End Sub

Private Sub CommandButton2_Click()
   On Error Resume Next
   Sim.Go
   If Err <> 0 Then
      MsgBox ("Error: " & Err.Description)
   Else
      [A5].Value = "Started"
   End If
End Sub

Private Sub CommandButton3_Click()
   On Error Resume Next
   Sim.Stop
   If Err <> 0 Then
```

Figure 20.3: The basic test client

The client is ready for a first test. Leave design mode (the left-most button on the Control Toolbox toolbar), click the ‘Init’ button and wait for the text “Launch successful’ to appear. You can also verify that the simulator is running by executing the `efoList` utility from the command line, see Section 14.7.1. Click the ‘Stop’ button in the Excel sheet to stop the simulator.
The Windows Application Event Log may give you a clue in case you encounter problems.

20.4.3 Adding a View

The example simulator has been build with the ‘External Simulator Access’ build option set in the Model Editor. This means that at simulator startup a `.exports` file, with the same basename as the simulator, will be searched for. This file specifies which nodes of the data dictionary will be available for reading and writing using so called ‘views’. More information can be found in the `extExport(3)` man page. The `Counter.exports` file looks like this:

```
/Counter readview  R
/Counter writeview W
```

This means that all variables in the /Counter node — the Counter.c file — are available for reading when a view is created with the name "readview" and they can be written when using a view that was created with the name "writeview". The following paragraphs describe how to create views in the MS Excel based client application.
Go to the VB editor and double click 'Module1' in the VBAProject tree. Add the following lines to the declarations:

```
Public ReadView As EuroSim.IvarView
Public dCounter As Double
```

Then add a 'Create View' button with the following code:

```
Private Sub CommandButton4_Click()
   On Error Resume Next
   Set ReadView = sim.CreateVarView ("readview")
   If Err <> 0 Then
      MsgBox ("Error: " & Err.Description)
   Else
      ReadView.AddDouble "dCounter", dCounter
      ReadView.Connect EuroSim.Read, 10
   End If
End Sub
```

That is all the code needed to have EuroSim copy the value of simulator variable 'dCounter' to the client variable 'dCounter'. The updates will occur at a frequency of 10 Hz.
Now we want to display the value of 'dCounter' in a cell of the sheet. We could add a button that invokes some code that copies the value of 'dCounter' into a cell, but there is a more sophisticated means to achieve this, which is described in the next paragraphs.

20.4.4 Receiving updates from the simulator

So far, the client has been calling the methods on the simulator interface of the EuroSim component. This is depicted in Figure 20.4.

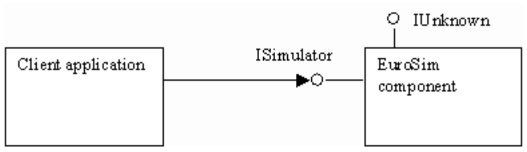


Figure 20.4: Client calling methods on the ISimulator interface

If the client application wants to keep track of changes in simulator variables, it could simply poll. However, if this is done from VBA code in, for example, an Excel spreadsheet, the complete Excel application would not be responsive to user input while polling. To solve this problem, the EuroSim COM interface provides an event callback mechanism.
Note that the client application has to implement an interface that the EuroSim component makes calls on. However, the EuroSim component specifies this interface in the type library. Since the component is the source of the calls on this outgoing interface, this interface is called a *source* interface. The client is called the *sink* for calls on this interface. The next paragraphs describe how to set-up a sink, or event handler, in VBA.

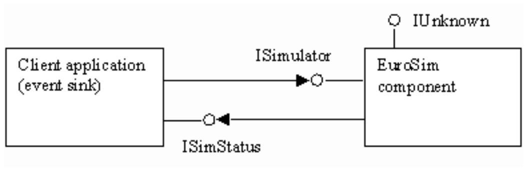


Figure 20.5: Component calling methods on the client interface

20.4.5 Creating an event handler in VBA

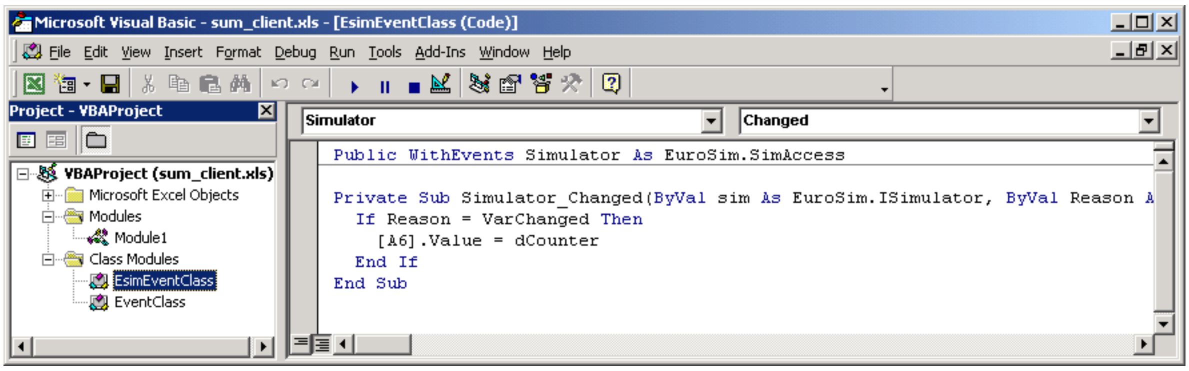
Right click the VBAProject tree and *Insert:Class* module. Select the new class and press F4 to display the properties window. Rename the class to ‘EsimEventClass’. Enter the following declaration in the code window of the new class:

```
Public WithEvents Simulator As EuroSim.SimAccess
```

This will declare an object named Simulator as an instance of the EuroSim.SimAccess class. The `WithEvents` statement tells VB that it receives events.
At the top of the code window, there are two drop down edit boxes. In the one on the left, select **Simulator** from the list. Since there is only one method, **Changed**, the VB editor automatically creates a subroutine called ‘Simulator_Changed’. Add the following line to this new subroutine:

```
If Reason = VarChanged Then
   [A6].Value = dCounter
End If
```

The above line of code writes the value of ‘dCounter’ to cell A6 on the sheet, each time the interface notifies our client that something has changed in the external simulator. Your VB editor should look similar to Figure 20.6.


Microsoft Visual Basic - sum_client.xls - [EsimEventClass (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Project - VBAProject
VBAProject (sum_client.xls)
Microsoft Excel Objects
Modules
Module1
Class Modules
EsimEventClass
EventClass
Simulator
Changed

```
Public WithEvents Simulator As EuroSim.SimAccess

Private Sub Simulator_Changed(ByVal sim As EuroSim.ISimulator, ByVal Reason A
  If Reason = VarChanged Then
    [A6].Value = dCounter
  End If
End Sub
```

Figure 20.6: Creating an event handler

We also need an instance of the event class: select ‘Module1’ and add the following line

```
Public EsimClass As New EsimEventClass
```

to the global declarations so that it looks like the code below:

```
Public sim As New EuroSim.SimAccess
Public ReadView As EuroSim.IvarView
Public dCounter As Double
Public EsimClass As New EsimEventClass
```

The last step is to install the sink. Go to the CommandButton1_Click subroutine and add the following line

```
Set EsimClass.Simulator = sim
```

so that it looks like the code below:

```
Private Sub CommandButton1_Click()
   On Error Resume Next
   sim.Launch "localhost", "C:\mysims\Counter", _
              "Counter.sim", "TestClient", 0
   If Err <> 0 Then
      MsgBox ("Error: " & Err.Description)
   Else
      [A5].Value = "Launch successful"
      Set EsimClass.Simulator = sim
   End If
End Sub
```

20.4.6 Sending updates to the simulator

This chapter will help you to modify your Excel application so that when you modify cells on your worksheet, these modified values are sent to the EuroSim simulator.
First, we need a view with write permissions. Add the following declarations to Module1:

```
Public WriteView As EuroSim.IvarView
Public newCounter As Double
```

Then add the following code to the CommandButton4_Click (CreateView button) subroutine:

```
Set WriteView = sim.CreateVarView("writeview")
If Err <> 0 Then
   MsgBox ("Error: " & Err.Description)
Else
   WriteView.AddDouble "dCounter", newCounter
   WriteView.Connect EuroSim.Write, 0
End If
```

The above code creates a relation between the local variable 'newCounter' and the simulator variable 'dCounter', which we monitor using the readview.
Excel Worksheet and Workbook level events are contained by the Worksheet and Workbook objects, respectively. However, there is no similar object to contain the Excel Application level events. Therefore you must use a Class Module to create an object that can accept and handle Application level events.
Open the VB Editor, and choose Class Module from the Insert menu to create a new Class Module. Select the class module and insert the following statement as a global declaration:

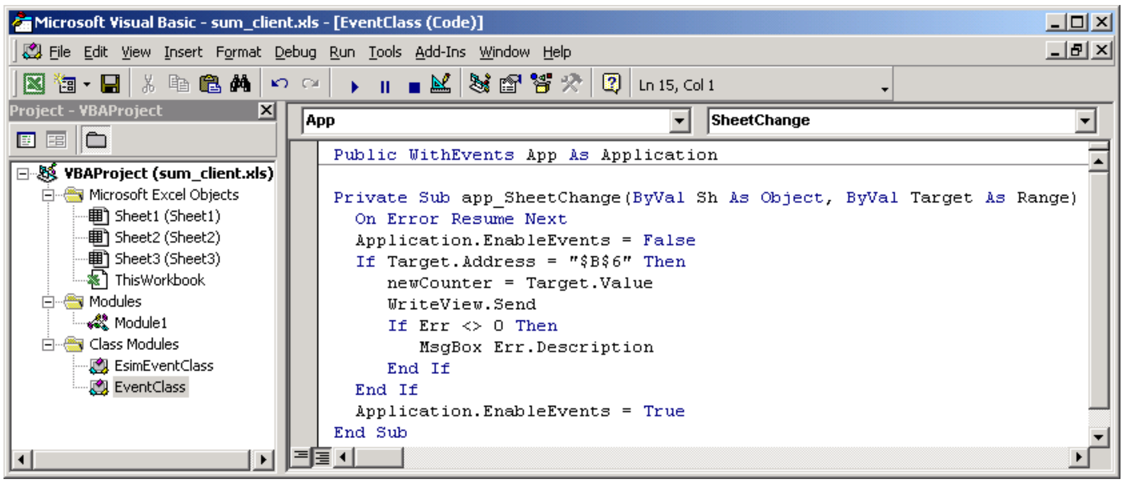
```
Public WithEvents App As Application
```

This will declare a variable named App as an instance of the Application class. The WithEvents statement tells Excel to send Application events to this object.
At the top of the code window, there are two drop down edit boxes. In the one on the left, select **App**, and in the one on the right, select the **SheetChange**. The VB editor will automatically insert the `Private Sub` and `End Sub` statements into the module. Add the following code to the app_SheetChange event handler:

```
On Error Resume Next
Application.EnableEvents = False
If Target.Address = "$B$6" Then
```

```
        newCounter = Target.Value
    WriteView.Send
        If Err <> 0 Then
            MsgBox Err.Description
        End If
    End If
    Application.EnableEvents = True
```

Each time cell B6 is changed, i.e. the user types a value, its value is copied to the variable called newCounter. This value is sent to the simulator variable 'dCounter' using the Send method on the WriteView object.
Press F4 to display the Properties window, and change the name of the class module to `EventClass`, see Figure 20.7.



```
Public WithEvents App As Application

Private Sub app_SheetChange(ByVal Sh As Object, ByVal Target As Range)
  On Error Resume Next
  Application.EnableEvents = False
  If Target.Address = "$B$6" Then
     newCounter = Target.Value
     WriteView.Send
     If Err <> 0 Then
        MsgBox Err.Description
     End If
  End If
  Application.EnableEvents = True
End Sub
```

Figure 20.7: The application event handler

Next, add the following line

```
    Public AppClass As New EventClass
```

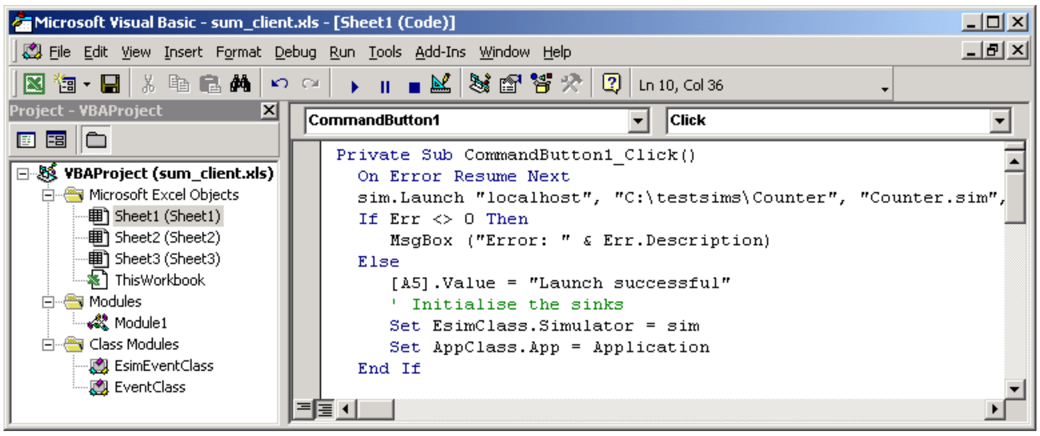
to the global declarations in Module1 so that it looks like the code below:

```
    Public sim As New EuroSim.SimAccess
    Public ReadView As EuroSim.IvarView
    Public WriteView As EuroSim.IvarView
    Public dCounter As Double
    Public EsimClass As New EsimEventClass
    Public AppClass As New EventClass
```

This will create an object called AppClass as a new instance of EventClass.
In order to receive application events, the App variable of the AppClass object must be set to the actual Excel application. One place to do this is in the CommandButton1_Click subroutine, using the following statement:

```
    Set AppClass.App = Application
```

Your VB editor should now look similar to Figure 20.8.
The MS Excel based client application is ready for another test. Leave design mode, launch the simulator, create the views and type a numeric value in cell B6. After pressing the `Enter` key, the application event handler will be called, which will send the value of cell B6 to the simulator.



```
Private Sub CommandButton1_Click()
  On Error Resume Next
  sim.Launch "localhost", "C:\testsims\Counter", "Counter.sim",
  If Err <> 0 Then
     MsgBox ("Error: " & Err.Description)
  Else
     [A5].Value = "Launch successful"
     ' Initialise the sinks
     Set EsimClass.Simulator = sim
     Set AppClass.App = Application
  End If
```

Figure 20.8: Setting the AppClass.App

Chapter 21

Web Interface reference

21.1 Introduction

This document describes how to setup and use the EuroSim Web Interface. The two main components of the web interface are

- the server
- the monitor
- the classes that describe the JAVA applet (client)

The server is the central component of the system. It communicates with client-side software (typically web browsers), with the monitor application, and with simulators (via the monitor).

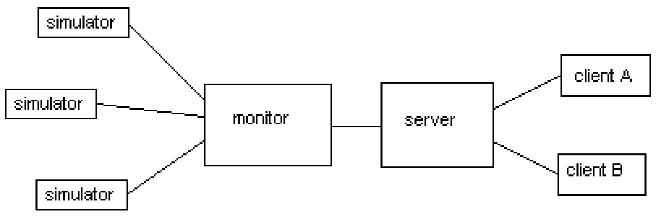


Figure 21.1: EuroSim Web Interface

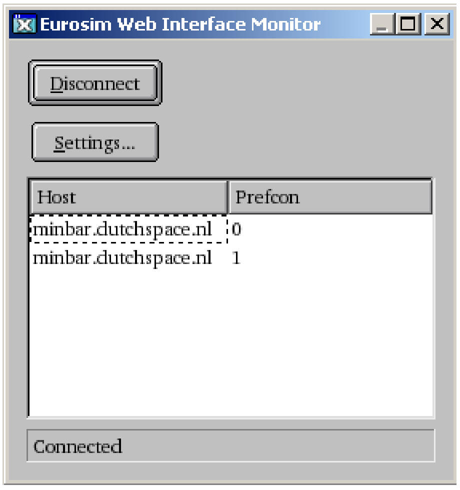
The server communicates with the clients using the HTTPS protocol (HTTP over SSL). The server uses the EuroSim protocol for communication with the simulators. Instead of letting the server connect directly to the simulator, the monitor sets up connections to the simulator and the server, after which it does nothing more than forward data between the two.
The monitor is installed on the same network as the simulators it has to watch. The server can request the monitor to scan its local network for simulators, and request a connection to a simulator.
How to use these applications is explained in more detail in the following chapters.

21.2 Monitor

This chapter explains how to use the monitor application.

21.2.1 User interface

The monitor has a simple graphical user interface (see Figure 21.2). It allows the user to connect to the server, disconnect from the server and to change the configuration. It also displays the state of the connection with the server and a list of connections to simulators.



Host	Prefcon
minbar.dutchspace.nl	0
minbar.dutchspace.nl	1

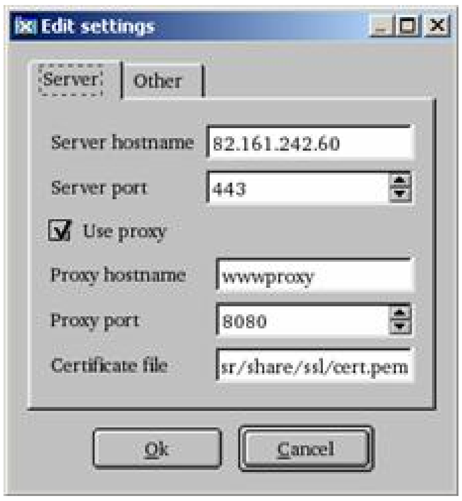
Connected

Figure 21.2: The Monitor GUI

By pressing the **Connect** button, the user initiates a connection to the server. This can be a direct connection, or via a proxy. The state of the connection is displayed on the bottom of the window. When the connection succeeds, the status message changes to 'Connected'.
On successful connection, the caption of the connect button changes to **Disconnect**. Pressing the button in this state closes the connection.
The **Settings** button brings up a configuration dialog, where the monitor settings can be adjusted. This is explained in more detail in the next section.
The listview below the buttons displays the simulator connections that are currently open.

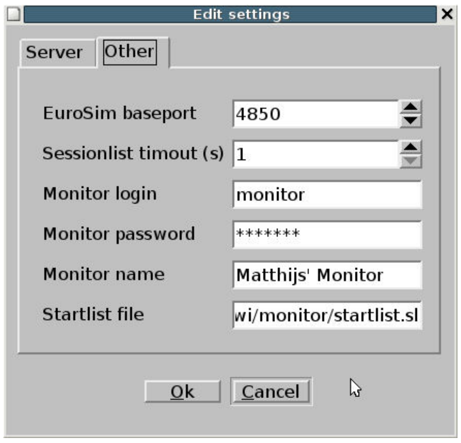
21.2.2 Settings

The settings dialog has two tabs where several important configuration parameters can be adjusted.
The 'Server hostname' is where the server can be found. This can be in the form of a hostname (for example 'www.dutchspace.nl', or an IP address in so called 'dotted decimal notation', as shown in Figure 21.3
.
The 'Server port' is the port number of the server. This is usually 443, the standard port for HTTPS (which is the protocol used by the server).
Next up are the proxy settings. If web access requires a proxy at the location where the monitor is installed, check the 'Use proxy' checkbox. This enables the 'Proxy hostname' and 'Proxy port' fields, which have the same meaning for the proxy as the 'Server hostname' and 'Server port' have for the server. The standard port for proxies is 8080.
The 'Certificate file' is the file that contains the certificates for 'Certificate Authorities'. On Linux systems, this typically is '/usr/share/ssl/cert.pem'. See Section 21.4 for a more detailed explanation of certificates.
The EuroSim baseport is normally 4850. This value gets added to the 'prefcon' value for simulator connections, to give the actual TCP port number.
The 'Monitor login' and 'Monitor password' are necessary to establish the connection to the server. Without a valid username and password it is not possible to use the web interface.
The 'Monitor name' is the name that appears in the monitor list that is sent to the client.
The 'Startlist file' is the path and filename to the file that describes the known simulators that can be started by the EuroSim Web Interface via this monitor.



Server hostname 82.161.242.60
Server port 443
Use proxy ☑
Proxy hostname wwwproxy
Proxy port 8080
Certificate file sr/share/ssl/cert.pem

Figure 21.3: The Monitor Settings (first tab)



EuroSim baseport 4850
Sessionlist timout (s) 1
Monitor login monitor
Monitor password *******
Monitor name Matthijs' Monitor
Startlist file wi/monitor/startlist.sl

Figure 21.4: The Monitor Settings (second tab)

The values of these settings are stored in the file $HOME/.qt/esimwebrc.

21.2.3 Startlist XML-file

Next to querying the local network for running simulators, the monitor also reads an xml-file to generate a startlist. The path to the startlist can be defined on the settings tab of a monitor.
An example of such a startlist file is given below.

```
<?xml version="1.0"?>
<startlist>
 <simulation>
  <id>Demo1</id>
  <name>Atos Origin Nederland Demo 1</name>
  <simfile>/home/nl27111/demo1/demo1.sim</simfile>
  <host>nwgesim002.nl.int.atosorigin.com</host>
 </simulation>
 <simulation>
  <id>Demo2</id>
  <name>Atos Origin Demo 2</name>
  <simfile>/home/nl27111/demo2/demo2.sim</simfile>
```

```
    <host>nwgesim002.nl.int.atosorigin.com</host>
  </simulation>
</startlist>
```

The file always has one startlist element, with one or more simulation elements. Every simulation has four child elements: id, name, simfile and host.
Note: The id field of a simulation has to be *one* word, without spaces.

21.3 Server

This chapter explains how to use the server application.

21.3.1 Startup

Starting the server can be done in 2 ways: via the command line, or via the internet daemon 'xinetd' (which is the preferred way).

21.3.1.1 Command line

When starting the server on the command line (for testing purposes), the option '—test' should be given. This makes the server listen on the port specified in the settings file, and sends all logging information to the console.

21.3.1.2 Using xinetd

The preferred way of running the server is via xinetd. This is a 'superserver' process that listens on the specified port on behalf of the server, and starts the server when there's an incoming connection on that port. The following configuration file could be used. Adjust the 'server' entry to point to the location where the server is installed, and copy the file to the /etc/xinetd.d/ directory.

```
service esimweb
{
  type = UNLISTED
  id = esimweb
  socket_type = stream
  user = root
  server = /usr/local/esimweb/server
  wait = yes
  protocol = tcp
  port = 443
  disable = no
}
```

21.3.1.3 Settings

Like the monitor, the configuration of the server is stored in a file in $HOME/.qt/esimwebrc. The table below lists the settings that can be adjusted in the file:

DefaultPage
The page that should be opened when the user does not request a specific page. Default is 'index.html'.

DocumentRoot
The root of the directory tree that contains the files that the user can browse.

ListenPort
: The tcp/ip port number that the server should listen on. This only has effect if the server is started with the '—test' option, otherwise it uses file descriptor 0. which it assumes to be initialized by xinetd to the file descriptor on which to accept a new connection.

PathToCertificate
: The path to the servers certificate file.

PathToPrivateKey
: The path to the servers private key file.

21.3.2 Authentication

The authentication information for clients and monitors is kept in a file called 'auth.xml' in the same directory as the server executable. It contains all valid user / password combinations.
Access control is divided into 2 'realms' (this term is used in the HTTP basic authorization scheme): one for clients, and one for monitors. The name of the client realm is "EuroSim Web Interface Client", and the name for the monitor realm is "EuroSim Web Interface Monitor". These names are hardcoded in the server, and should match exactly.
Below is an example of such an authentication file:

```
<authinfo>
 <realm name="EuroSim Web Interface Client">
  <user login="eurosim" password="hard2guess"/>
  <user login="johndoe" password="2hard4u"/>
 </realm>
 <realm name="EuroSim Web Interface Monitor">
  <user login="demo" password="xyz123"/>
 </realm>
</authinfo>
```

This file would give access to 2 clients: one with username 'eurosim' and password 'hard2guess', and one with username 'johndoe' and password '2hard4u'. Access is also granted for a monitor with username 'demo' and password 'xyz123'.

21.4 Certificates

This chapter will try to explain the basics of certificates.

21.4.1 What is a certificate?

(This section was taken from the QtSSLSocket documentation)
A certificate is a document which describes a network host's identity. It contains, among others, the DNS name of the host, the name and ID of the certificate issuer, an expiry date and a digital signature.
Certificates are created together with a host's private key. The certificate is either self-signed or signed by a certification authority (CA). Safe communication requires the certificate to be signed by a CA. Basically, a self-signed certificate can never be used to verify the identity of a server, but it can be used to seed the ciphers used to encrypt communication. For this reason, self-signed certificates are often used in test systems, but seldom in production systems.
Official CAs sign public certificates for a certain price. Two well-known official CAs are Thawte (http://www.thawte.com/) and Verisign (http://www.verisign.com/). To obtain a CA signed certificate, a "certificate request" (unsigned certificate) is generated and posted to certain forms on the CAs' home pages.
It is quite possible to set up one's own local CA and use that to sign servers' certificates. Although this avoids the expense of using an official CA, all clients must then have a local copy of your own CA's SSL certificate.

21.4.2 Creating a self-signed certificate

It is possible to use the openssl utility to create a self-signed certificate and the corresponding private key. Of course this is only useful for testing purposes. Use the following command:

```
openssl.exe req --x509 -newkey rsa:1024 -keyout server.key -nodes -days
365 -out server.crt
```

This creates a 1024 bit RSA private key, and a certificate that is valid for 365 days. Make sure the server can find these files by specifying their locations in the configuration file.

21.5 JAVA applet interface

This chapter describes the JAVA applet of the EuroSim Web Interface.

21.5.1 Start screen

When visiting the main URL for the web interface, you will probably be presented with a warning about the servers certificate. This is because at this moment, the server uses a self-signed certificate, instead of one issued by a genuine certificate authority (CA). For the moment, this warning can be ignored.
Note: To run an applet it is necessary to have a JAVA Virtual Machine (JVM) installed and enabled in your browser.
After this, the JAVA applet will be presented (see Figure 21.5).

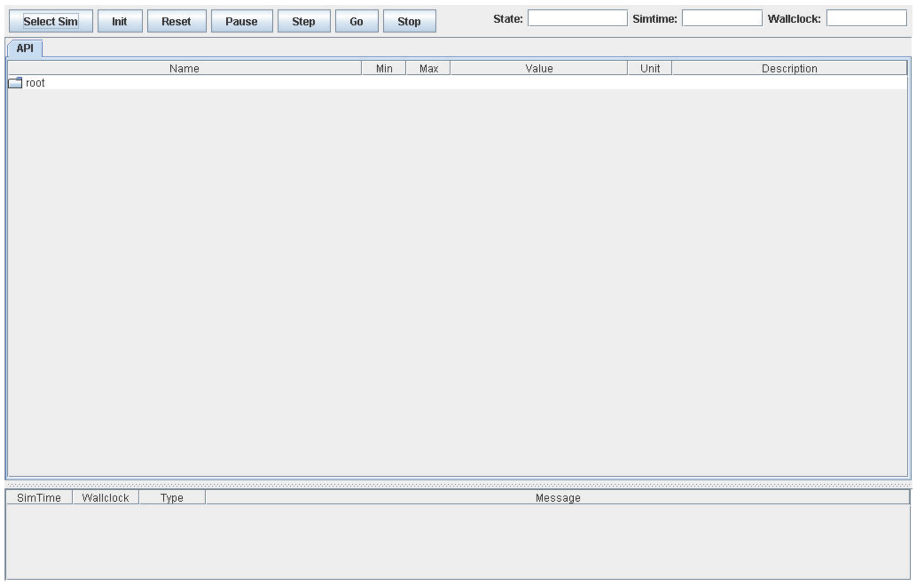


Figure 21.5: Java applet start screen

21.5.1.1 Control buttons

The control buttons are located on top of the screen. These buttons have the same functionality as the buttons on the toolbar of the Simulation Controller.

21.5.1.2 Status information

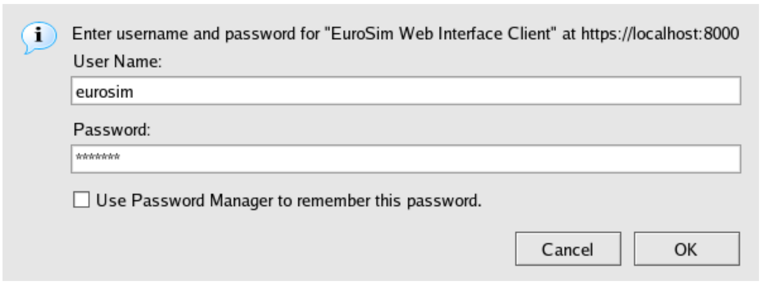
Displayed next to the control buttons are three fields with status information.
At first the current state of the simulator, second the simulation time and third the wall clock time.

21.5.1.3 Message window

Located at the bottom of the screen is the message pane. On the message pane all messages are displayed. This includes messages generated by the simulator (e.g. when starting the simulator, or when pausing it), errors from the scheduler.

21.5.2 Select Simulator

Clicking on the button 'Select Sim' will pop-up a dialog with a list of available monitors. Before this list is shown however, it is necessary that you login using a username and a password (see Figure 21.6).



Enter username and password for "EuroSim Web Interface Client" at https://localhost:8000

User Name: eurosim

Password: *******

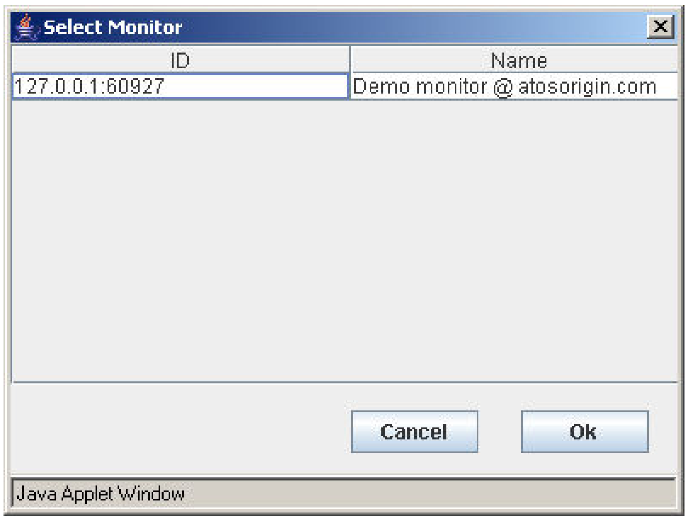
☐ Use Password Manager to remember this password.

Figure 21.6: Login dialog

When your input is accepted, you will be taken to the monitor list. Otherwise, the login dialog will keep asking you for you credentials. Pressing **Cancel** will stop this, resulting in a '401 Unauthorized' message.

21.5.3 Monitor list dialog

After you have successfully logged in, you will see a dialog as shown in Figure 21.7.



Select Monitor

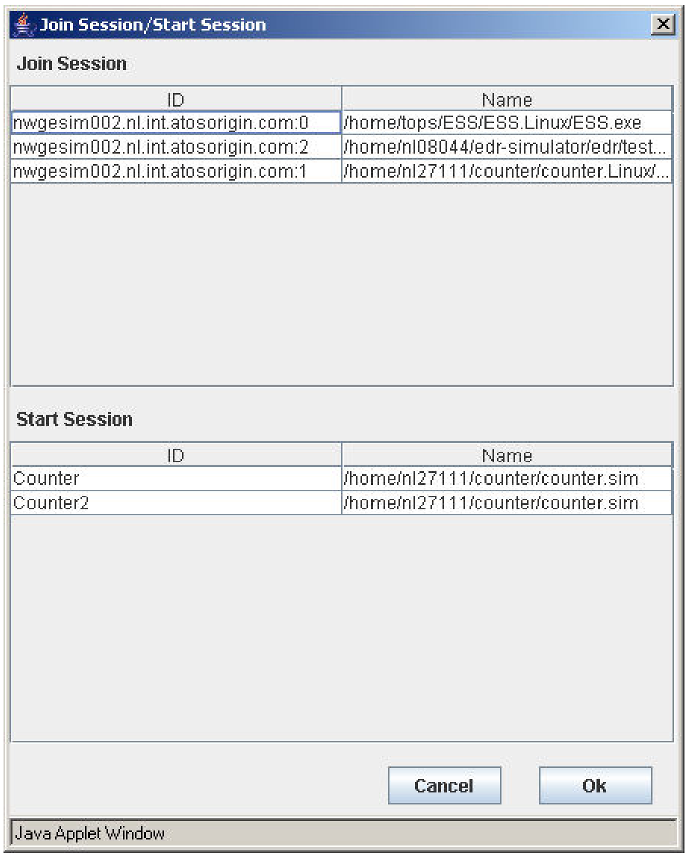
ID	Name
127.0.0.1:60927	Demo monitor @ atosorigin.com

Figure 21.7: Monitor list dialog

This shows a list of all monitors that are currently connected to the server. To retreive the sessionlist/s-tartlist of a monitor, select a row and click 'Ok'.

21.5.4 Session list dialog

The session list dialog looks like Figure 21.8.



ID	Name
nwgesim002.nl.int.atosorigin.com:0	/home/tops/ESS/ESS.Linux/ESS.exe
nwgesim002.nl.int.atosorigin.com:2	/home/nl08044/edr-simulator/edr/test...
nwgesim002.nl.int.atosorigin.com:1	/home/nl27111/counter/counter.Linux/...

ID	Name
Counter	/home/nl27111/counter/counter.sim
Counter2	/home/nl27111/counter/counter.sim

Figure 21.8: Session list dialog

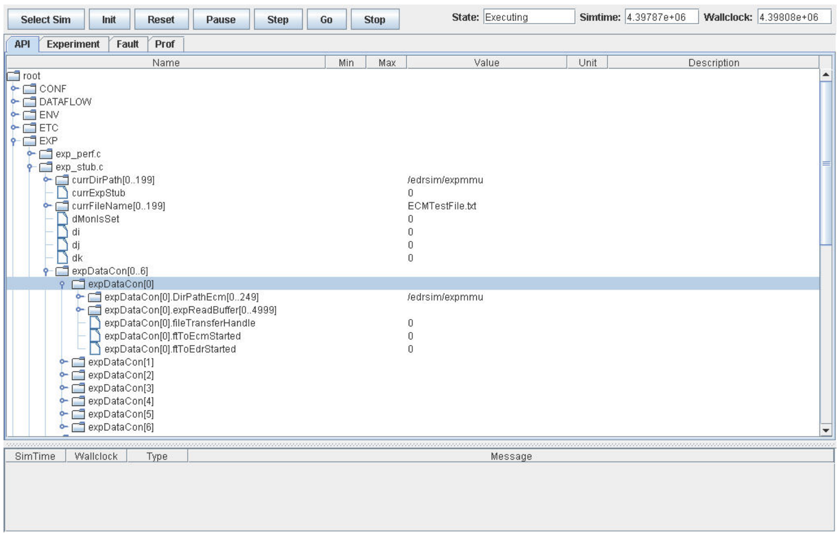
It shows a list of all sessions that are currently running on the monitors local network and a list of simulators that can be started. Each running session is represented on a row in the upper table that contains its hostname, prefcon number and the name and path of the simulator executable. The lower table contains a short name and the name and path for sessions that can be started.
Join or start a session by selecting the row and pressing the **Ok** button.

21.5.5 API Tab

After joining or starting a session the JAVA applet fills the API tab with all the variables and the tab will look like Figure 21.9.
The API tab page is a Dictionary Browser with some extra functionality. When no simulation is running it just shows the dictionary with a few extra columns to show the minimum and maximum values, the unit of the value, and the description of the variable.
As long as a connection to the simulator is active this column will show the current value of that variable just like a monitor in an MMI tab page. By clicking on the value you can edit it and set the variable to a new value.

21.5.6 MMI Tab

When the applet is finished filling up the API tab with variables, the applet generates the MMI (Man Machine Interface) tabs as they were designed in the Simulation Controller.
An example of a MMI tab is given in Figure 21.10
A MMI tab page is a canvas on which monitors are displayed to monitor variables in the simulation.
There are two basic types of monitors: alpha numerical, i.e. each variable is presented as a caption followed by the value, and the graph monitor, where each variable is tracked over time (or possibly

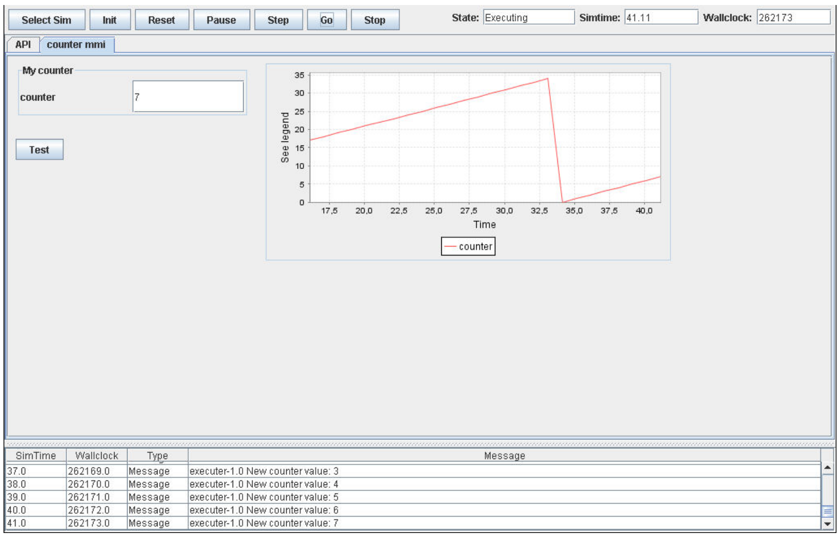


State: Executing Simtime: 4.39787e+06 Wallclock: 4.39808e+06

Name	Min	Max	Value	Unit	Description
root					
CONF					
DATAFLOW					
ENV					
ETC					
EXP					
exp_perf.c					
exp_stub.c					
currDirPath[0..199]			/edrsim/expmmu		
currExpStub			0		
currFileName[0..199]			ECMTestFile.txt		
dMonIsSet			0		
di			0		
dj			0		
dk			0		
expDataCon[0..6]					
expDataCon[0]					
expDataCon[0].DirPathEcm[0..249]			/edrsim/expmmu		
expDataCon[0].expReadBuffer[0..4999]					
expDataCon[0].fileTransferHandle			0		
expDataCon[0].ftToEcmStarted			0		
expDataCon[0].ftToEdrStarted			0		
expDataCon[1]					
expDataCon[2]					
expDataCon[3]					
expDataCon[4]					
expDataCon[5]					
expDataCon[6]					

SimTime	Wallclock	Type	Message

Figure 21.9: API tab



State: Executing Simtime: 41.11 Wallclock: 262173

My counter

counter 7

SimTime	Wallclock	Type	Message
37.0	262169.0	Message	executer-1.0 New counter value: 3
38.0	262170.0	Message	executer-1.0 New counter value: 4
39.0	262171.0	Message	executer-1.0 New counter value: 5
40.0	262172.0	Message	executer-1.0 New counter value: 6
41.0	262173.0	Message	executer-1.0 New counter value: 7

Figure 21.10: MMI tab

against another variable) and plotted on a graph. Besides monitoring variables you can also have Action Buttons to execute MDL scripts or to enable/disable recorders or stimuli.

21.5.6.1 Alpha numerical monitors

Alpha numerical monitors display a window in the MMI tab page in which the current value of one or more variables will be presented. These values will be updated every second.

21.5.6.2 Graphical monitors

Graphical monitors use one of three types of graphs to display the values of variables:

XY Plot One or more variables against an independent variable.

Simulation Time
Plot one or more variables against the simulation time.

Wall Clock Time
Plot one or more variables against the wall clock time.

21.5.6.3 Action buttons

Action buttons are used to execute MDL scripts or to enable/disable recorders or stimuli.

21.6 Reference

This chapter provides a reference to the methods of the server interface, and a description of the XML formats that are used.

21.6.1 Server interface

The following sections describe how to call the methods of the server interface from clients. Method calls are performed by requesting a URL with the method name and parameters encoded in it. For example, to request the monitor list from a server located at www.hostname.com, the following URL is used:
https://www.hostname.com/esim?method=getMonitorList

Additional parameters are encoded the same way, for example:
https://www.hostname.com/esim?method=getSessionList&monitorId=localhost:0

The result format can be specified by the format parameter. It can either be ‘xml’ or ‘html’, and defaults to ‘xml’. The html version is of course better suited for a web browser interface, while the xml version will probably be used more from scripts. An example of requesting the session list in html:
https://www.hostname.com/esim?method=getSessionList&monitorId=localhost:0&format=html

21.6.1.1 Retrieving the monitor list

Clients can request a list of the monitors that are currently connected to the server. This list contains the id and the name of the monitors. The monitor id is particularly useful, since it is used in subsequent calls to refer to the monitor.
This method cannot fail.

Method	**getMonitorList**	
Parameters	None	-
Return	A monitor list on success, or an error if something went wrong.	

Example:
To request the monitor list from a server at address hostname, use the following URL:
https://hostname/esim?method=getMonitorList

21.6.1.2 Retrieving the session list

The session list is a list of session-info structures of simulators that are running on the same local network as the monitor. It contains parameters of each session, such as simulator name, path of the data dictionary, etc. If the monitor is able to read the dictionary, each session-info also contains a list of variables of the simulator. Because the session list is specific for a certain monitor, it is necessary to pass the monitor id to the method.
This method could fail if the specified *monitorId* is unknown.

Method	**getSessionList**	
Parameters	monitorId	The id of the monitor whose session list is requested.
Return	A session list on success, or an error if something went wrong.	

Example:

To request the session list for a monitor with id localhost:0, from a server at address hostname, use the following:
https://hostname/esim?method=getSessionList&monitorId=localhost:0

21.6.1.3 Retrieving the view list

The server keeps a list of views for each user. The user is tracked by the server using cookies containing a session id. The view list contains for each defined view a list of variables and their values, and the simulator state, simtime and runtime.
This method cannot fail.

Method	**getViewList**	
Parameters	None	-
Return	A view list on success, or an error if something went wrong.	

Example:
To request the list of views currently in your session at server hostname, use the following URL:
https://hostname/esim?method=getViewList

21.6.1.4 Adding a view

The user can add a view to the view list by using the addView method. The name of the new view is specified by the 'viewId' parameter. The 'monitorId' and 'simId' parameters are used to identify the simulator for which the view is constructed.
This method could fail if any of the specified ids are unknown, or if the user has already defined another view with the same name.

Method	**addView**	
Parameters	monitorId	The id of the monitor.
	simId	The id of the simulator .
	viewId	The name of the view that is to be created.
Return	A viewlist on success, or an error if something went wrong.	

Example:
To add a view 'DemoView' to the simulator with id 'sim:0' on monitor 'localhost:0', use the following URL:
https://hostname/esim?method=addView&monitorId=localhost:0&simId=sim:0&viewId=DemoView

21.6.1.5 Deleting a view

Views can also be deleted from the view list. This is done using the delView method, which takes the same parameters as the addView method discussed above.
This method could fail if any of the ids are unknown.

Method	**delView**	
Parameters	monitorId	The id of the monitor
	simId	The id of the simulator
	viewId	The name of the view that is to be created
Return	A viewlist on success, or an error if something went wrong.	

Example:
To delete a view 'DemoView' on the simulator with id 'sim:0' on monitor 'localhost:0', use the following URL:
https://hostname/esim?method=delView&monitorId=localhost:0&simId=sim:0&viewId=DemoView

21.6.1.6 Adding a variable

Adding variables to a view is done using the addVariable method. This method takes 4 parameters: the monitorId, simId and viewId have the same meaning as above, and the varName parameter contains the name of the variable that is to be added.
This method could fail if any of the ids (*monitorId*, *simId* and *viewId*) are unknown, or if the view already contains a variable with the specified name.

Method	**addVariable**	
Parameters	monitorId	The id of the monitor
	simId	The id of the simulator
	viewId	The name of the view where the variable has to be added
	varName	The name of the variable to be added
Return	A view list on success, or an error if something went wrong.	

Example:
To add a variable 'Altitude' to view 'DemoView' for simulator 'sim:0' on monitor 'localhost:0', use the following URL:
https://hostname/esim?method=addVariable&monitorId=localhost:0&simId=sim:0&viewId=DemoView&varName=Altitude

21.6.1.7 Deleting a variable

Deleting a variable from a view is done using the delVariable method. It takes the same parameters as the addVariable method above.
This method could fail if any of the specified ids are unknown.

Method	**delVariable**	
Parameters	monitorId	The id of the monitor
	simId	The id of the simulator
	viewId	The name of the view where the variable has to be deleted
	varName	The name of the variable to be deleted
Return	A view list on success, or an error if something went wrong.	

Example:
To delete the variable 'Altitude' from view 'DemoView' for simulator 'sim:0' on monitor 'localhost:0', use the following URL:
https://hostname/esim?method=delVariable&monitorId=localhost:0 &simId=sim:0&viewId=DemoView&varName=Altitude

21.6.2 XML formats

This section contains DTD and examples for all XML formats used in the web interface.

21.6.2.1 The monitor list

The monitor list is a structure that contains multiple monitor elements, all consisting of an id and a name element.

Format:

```
<!ELEMENT id (#PCDATA)>
<!ELEMENT monitor (id, name)>
<!ELEMENT monitorlist (monitor*)>
<!ELEMENT name (#PCDATA)>
```

Example:

```
<?xml version="1.0"?>
<monitorlist>
 <monitor>
  <id>127.0.0.1:36506</id>
  <name>Demo monitor {\@} atosorigin.com</name>
 </monitor>
</monitorlist>
```

21.6.2.2 The session list

The session list structure contains multiple session elements, all consisting of a hostname, prefcon and simulator element.

Format:

```
<!ELEMENT hostname ({\#}PCDATA)>
<!ELEMENT prefcon ({\#}PCDATA)>
<!ELEMENT session (hostname, prefcon, simulator)>
<!ATTLIST session
 simid CDATA {\#}REQUIRED
>
<!ELEMENT sessionlist (session*)>
<!ATTLIST sessionlist
 monitorid CDATA {\#}REQUIRED
 monitorname CDATA {\#}REQUIRED
>
<!ELEMENT simulator ({\#}PCDATA)>
```

Example:

```
<?xml version="1.0"?>
<sessionlist monitorid="127.0.0.1:36506" monitorname="Demo monitor @ foobar.c
<session simid="demo.foobar.com:0">
<hostname>demo.foobar.com</hostname>
<prefcon>0</prefcon>
<simulator>/home/demo/foo/ESS.Linux/ESS.exe</simulator>
</session>
<session simid="demo.example.com:0">
<hostname>demo.example.com</hostname>
<prefcon>0</prefcon>
<simulator>/home/test/bar/xyz.Linux/xyz.exe</simulator>
</session>
</sessionlist>
```

21.6.2.3 Sessioninfo

The session info structure contains session parameters (like the dictionary path, working directory, etc) and a list of available variables.

Format:

```
<!ELEMENT description (#PCDATA)>
<!ELEMENT dict (#PCDATA)>
<!ELEMENT exports EMPTY>
<!ELEMENT gid (#PCDATA)>
<!ELEMENT hostname (#PCDATA)>
<!ELEMENT initconds (item)>
<!ELEMENT item (#PCDATA)>
<!ELEMENT max (#PCDATA)>
<!ELEMENT min (#PCDATA)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT pid (#PCDATA)>
<!ELEMENT prefcon (#PCDATA)>
<!ELEMENT recorderdir (#PCDATA)>
<!ELEMENT scenarios (item+)>
<!ELEMENT schedpath (#PCDATA)>
<!ELEMENT sessioninfo (hostname, simpath, workdir, simulator, schedpath,
dict, model, recorderdir, exports, initconds?, scenarios?, prefcon, uid,
gid, pid, variables)>
<!ATTLIST sessioninfo
 simid CDATA #REQUIRED
 monitorid CDATA #REQUIRED
>
<!ELEMENT simpath (#PCDATA)>
<!ELEMENT simulator (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT uid (#PCDATA)>
<!ELEMENT unit (#PCDATA)>
<!ELEMENT var (name, type, unit, min, max, description)>
<!ELEMENT variables (var+)>
<!ELEMENT workdir (#PCDATA)>
```

Example:

```
<?xml version="1.0"?>
<sessioninfo simid="demo.foobar.com:0" monitorid="127.0.0.1:36506">
 <hostname>nwgesim002.nl.int.atosorigin.com</hostname>
  <simpath>/home/demo/foo/ESS.sim</simpath>
  <workdir>/home/demo/foo</workdir>
  <simulator>/home/demo/foo/ESS.Linux/ESS.exe</simulator>
  <schedpath>/home/demo/foo/ESS.sched</schedpath>
  <dict>/home/demo/foo/ESS.Linux/ESS.dict</dict>
  <model>/home/demo/foo/ESS.model</model>
  <recorderdir>/home/demo/foo/2005-02-18/12:09:37</recorderdir>
  <exports/>
  <initconds>
   <item>/home/demo/foo/ESS.init</item>
  </initconds>
  <scenarios>
```

```
    <item>/home/demo/foo/Prof.mdl</item>
    <item>/home/demo/foo/Etc.mdl</item>
    <item>/home/demo/foo/Fault.mdl</item>
    <item>/home/demo/foo/Rec.mdl</item>
  </scenarios>
  <prefcon>0</prefcon>
  <uid>1005</uid>
  <gid>1005</gid>
  <pid>14686</pid>
  <variables>
  <var>
    <name>speed</name>
    <type>int</type>
    <unit>m/s</unit>
    <min>0</min>
    <max>100</max>
    <description>The speed of the object</description>
  </var>
  <var>
    <name>acceleration</name>
    <type>int</type>
    <unit>m/s2</unit>
    <min>-10</min>
    <max>10</max>
    <description>The acceleration of the object</description>
  </var>
 </variables>
</sessioninfo>
```

21.6.2.4 The view list

The view list structure contains multiple view elements, all consisting of a name, simstate, simtime and runtime element, and a list of variables.

Format:

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT runtime (#PCDATA)>
<!ELEMENT simstate (#PCDATA)>
<!ELEMENT simtime (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT var (name, value)>
<!ELEMENT variables (var*)>
<!ELEMENT view (name, simstate, simtime, runtime, variabless)>
<!ATTLIST view
monitorid CDATA #REQUIRED
simid CDATA #REQUIRED
>
<!ELEMENT viewlist (view)>
```

Example:

```
<?xml version="1.0"?>
<viewlist>
 <view monitorid="127.0.0.1:36506" simid="demo.example.com:0">
```

```
  <name>DemoView</name>
  <simstate>Executing</simstate>
  <simtime>6.90288e+06</simtime>
  <runtime>6.90307e+06</runtime>
  <variables>
   <var>
    <name>ball{\_}{\_}height</name>
    <value>123.456</value>
   </var>
   <var>
    <name>ball{\_}{\_}velocity</name>
    <value>3.1415</value>
   </var>
  </variables>
 </view>
</viewlist>
```

21.6.2.5 Errors

Errors can occur for a number of reasons, for example because a specified id (monitorId, viewId, simId) is unknown, or because an addVariable command is issued for a variable that is already present in the view. Errors simply contain a message about what went wrong.

Format:

```
<!ELEMENT error (message)>
<!ELEMENT message (#PCDATA)>
```

Example:

```
<?xml version="1.0"?>
<error>
 <message>An unknown error occurred</message>
</error>
```

Part IV

Appendices

Appendix A

Abbreviations

ADF	Application Definition File
AFAP	As Fast As Possibe
API	Application Programmers Interface
ASCII	American Standards Code for Information Interchange
COTS	Commercial Off The Shelf
Dict	Dictionary
EFO	EuroSim Follow-On
EI	External Interrupt
ERA	European Robotic Arm
ESA	European Space Agency
Esim	EuroSim
ESTEC	European Space Research and Technology Centre
EuroSim	European Real-time Operations Simulator
F77	Fortran 77
FFT	Fast Fourier Transform
DS	Dutch Space
GNAT	GNU ADA Translator
GUI	Graphical User Interface
HIL	Hardware In the Loop
HTML	HyperText Markup Language
Hz	Hertz
ID	Identification
I/O	Input/Output
LCM	Least Common Multiple
MDL	Mission Definition Language
MIF	Maker Interchange Format
Mk	Mark
MMI	Man-Machine Interface
NIVR	Netherlands Agency for Aerospace Programs
NLR	National Aerospace Laboratory NLR

org	Organization
OSF	Open Software Foundation
RCS	Revision Control System
SGI	Silicon Graphics Incorporated
SMDL	Simulation Model Definition Language
SMI	Simulation Model Interface
SMP	Simulation Model Portability
SMP2	Simulation Model Portability 2
SPR	Software Problem Report
SUM	Software User Manual
XML	Extensible Markup Language

Appendix B

Definitions

Action From a user's perspective, an action is part of his scenario, and defines both the required response to be taken when an *event* occurs, plus the required event. An action can be one of stimulus, recorder, monitor, intervention, and event. EuroSim provides specific editors for default recorders and stimuli, and a generic action editor for all other actions and customized recorders, stimuli and monitors. Monitor actions are obsolescent and are replaced with MMI definitions.

Application definition file
Format of files created by LynX; contain initialization and run-time information for a Vega application. Files have a .adf extension.

Data dictionary
A list of public data variables and parameters extracted from *model* code, i.e. those which are accessible to the user for (optionally) updating, monitoring and recording. The list is augmented with descriptive information (such as units, default values, ranges).

Data View
A subset of the data items in the EuroSim *data dictionary*. Used to define data items which are to be read/written by an *external simulator* at run time, and therefore provides a mechanism for sharing data between two independent *simulators*.

Entrypoint
A function or procedure in the model code (for which some restrictions apply) which can be used to create *tasks* in the Schedule Editor.

Event A discrete occurrence during a *simulation run*, which (can) cause a change in the behavior of the system being simulated, for example a component failure.

Execution state
The state of a *simulator*. Certain user requests are only valid in certain states.

External simulator
A simulator which is not running under EuroSim.

Facility management
The means of providing maintenance support and project and user management during the simulation life cycle.

Flight format
Binary format used for input and output by the MultiGen and ModelGen database modelling tools. It is a comprehensive format that can represent nearly all imaging concepts. Files in Flight format are structured as a linear sequence of records and have a .flt extension.

Hardware-in-the-loop
A piece of equipment which forms part of the real-world system, which is given a real-time interface to the simulation loop.

Initial condition
: Consistent set of model state values, to put the *model* into a particular state at the beginning of a *simulation run*. In EuroSim, the initial condition can be created with the Initial Condition Editor, or it can be a snapshot of values from previous simulation run.

Journal Information resulting from a particular *simulation run*(excluding sampled data values), e.g. log of executed *event* s, error/warning messages, and *marks*.

Man-in-the-loop
: A person taking on the role of an operator within the real-world application, who is provided with a real-time interface to the simulation loop.

Mark A pointer or reference mark made by a user during a *simulation run*, to provide an easy means of returning to a point of interest during *test analysis*.

Simulation Definition
: Complete definition of a particular test for a particular *model* and *schedule*, specifying the *initial conditions*, *stimuli* and variables which are to be recorded. For on-line evaluation, variables can also be viewed on screen by specifying monitors.

MMI *Definition*
: Defines the contents of a tab page in the Simulation Controller used for interacting with the simulator. Normally this tab page contains one or more monitors.

Model A set of components (*sub-models* and data files) which together define the data and behavioral characteristics of a specific real-world system, or part thereof. See *Simulator*.

Observer
: The user who (optionally) attends a*simulation run* and who may select variables for viewing, and *mark* interesting observations, but who is not able to affect the execution outcome in any way.

Operational modes
: EuroSim provides different modes of use which are available to one or more users; for example, the Model Developer uses EuroSim for *simulator* development, the Test Analyst uses it for analysis of *test results*. Particular user activities are only available during particular modes, for example *application model* s can only be updated during *simulator development*. EuroSim is able to support two or more modes simultaneously. See *simulator development*, *test preparation*, *test execution*, and *test analysis*.

Phase A time offset between completion of one *task* and activation of another task which is dependent on that completion, defined as a quantity of wall-clock time.

Real-time
: During real-time execution or interfacing, the time-lining of the activities appears to be that which would be seen in an equivalent situation in the real-world. This is achieved through guaranteed periodicity of processing and response time within fixed deadlines.

Schedule
: A set of attributed tasks, timers, scheduling events and their respective dependencies. The overall behavior of a schedule is deterministic, whereas that of a single task need not be. A schedule is executed by the scheduler. The scheduler has four states: Initializing, Standby, Executing and Exiting. Every state has its own schedule. The same task may appear in one or more state schedules.

Simulation
: The process of using *models* that behave or operate like a given system when provided a set of controlled inputs.

Simulation program
: The computer program, built out of *simulator software*, used for the simulation.

Simulation run
Execution of a *simulator* according to specified *simulation definition*.

Simulator
A hardware device or *simulation program* or combination of both with which *simulation* can be performed. A simulator together with a *simulation definition* can be used to start a *simulation run*.

Simulator development
Mode of operation, where the Model Developer can create and/or update *model*s and *simulator* definitions, and generate simulators. See *Operational Modes*.

Simulator software
Model-dependent software combined with model-independent software for the performance and control of real time and non-real time *simulation*.

State The current phase in the execution of the simulation. EuroSim states are: initialization, standby, executing and exit.

Stimuli A set of data which are input to the *model* during a *simulation run*, which represent data from an interfacing system or sub-system which would normally be present in the real-world; they can be used during replays of simulation runs, to provide copies of the original operator inputs.

Sub-model
A component of a *model*, which defines (in source code) an element or set of elements within the real-world application. The parts of a sub-model visible to other “users” are the set of accessible state data items (which are listed as part of the model *data dictionary*) and a set of operations which can be called by other sub-models or listed within a *task* within the schedule.

System services
A set of services offered by EuroSim which can be called directly from *model* code, for example in order to request information on the current simulation (e.g. simulated time, *execution state*), or to communicate with HIL devices.

Task A unit within a model schedule consisting of an ordered list of one or more *entrypoints*. Task execution starts with the first entrypoint listed, and suspends (always) after the last entrypoint listed has been executed. It is possible for tasks to be executed in parallel in a multi-processor environment.

Test analysis
Mode of operation, where the Test Analyst can mathematically analyze *test results,* replay visual images and export data for external use. See *Operational modes*.

Test Conductor
The user who operates the *simulator* as a tool to perform a *simulation run*.

Test execution
Mode of operation, where the *Test Conductor* has interactive control of a *simulation run*, and may initiate on-line events. The Test Conductor and (optionally) an *Observer* may also monitor *data dictionary* item values and create *marks*. See *Operational modes*.

Test preparation
Mode of operation, where the *Test Conductor* can create and/or update *simulation definitions*, and an *Observer* can identify *data dictionary* items for monitoring. See *Operational modes*.

Test results
All information resulting from a particular *simulation run*, i.e. the *journal* and the recorded *data dictionary* item values.

Appendix C

Scheduler Errors

In this appendix, two categories of errors are described:

- Errors generated by the Schedule Editor when creating or modifying a schedule.
- Errors generated by the EuroSim scheduler during simulation runs.

C.1 Schedule Editor errors

The Schedule Editor helps the model developer by indicating where problems arise during schedule definition. When one of the items placed on the schedule view is red, then there is an error for that item. The error can be viewed in the item attributes window. Below the possible error messages are described.

`name_unique`
: The name entered is already in use by another task. Change the name.

`number_of_input_flows`
: The item needs (mandatory) input. Add an input.

`number_of_output_flows`
: The item needs (mandatory) output. Add an output.

`active_flows`
: There is no active flow. Active flows are flows from data generating items. Connect a data generating item.

`frequency_mismatch`
: A task has two input with different input frequencies, or a synchronous store has an input frequency which does not match the assigned input frequency. Remove one of the inputs, change the frequency of one of the inputs, or use a synchronous store in one of the flows.

`frequency_zero`
: The timer of the synchronous store has a frequency of zero. Change it.

`incorrect_ratio`
: The ratio of the input and output frequency of a synchronous store is not *1:n* or *n:1*. Adjust one or both of the frequencies.

`cycle`
: There is a cycle in the schedule (i.e. following the flows you can come back where you started). Break the cycle by removing a flow or task.

`critical`
: Timing problem. The scheduler can not guarantee that the task can be completed in the available time. Modify timing of item or items connected to item concerned.

C.2 Scheduler run-time messages

The errors in this section are generated by the EuroSim scheduler during a simulation run. Each error has in the margin one or two of the following symbols:

N This message is an informational message only. No action is required.

W This message is a warning. It indicates a potential problem, which does not yet prevent the system proceeding.

E This message is an error. The system cannot proceed.

S This message should not occur (it stems from a file generated by EuroSim itself). Submit an SPR for this message.

Each message is accompanied by a short description and recovery suggestions if recovery by the user is possible.

ES: `at line` *`nnn`*`, syntax error`
This error is flagged when the textual schedule definition file contains a syntax error.

ES: `cannot open scheduler description file` *`sss`*
The schedule definition file could not be opened at initialization of `RT_SCHD`, probably because it is not present in the current directory. Make sure that the schedule definition in a file named `SCHEDULE_FILE` is present in the current directory, and restart `RT_SCHD`.

E: `at line` *`nnn`*`, number of real time processors must be within the range [1...p]`
The schedule definition file requests a number of real time processors which is larger than physically available. Correct the definition file by choosing a processor in the reported range, or restart with real time privileges off (no super-user authorities). This latter will result in non real-time execution mode, in which any number of 'real time' processors may be emulated.

E: `at line` *`nnn`*`, basic frequency must be within the range (0...f]`
A scheduler clock frequency beyond the system-imposed limit has been requested in the schedule definition file. Choose a clock frequency which falls within the reported range.

ES: `at line` *`nnn`*`, task` *`sss`* `has not been defined in the current state`
In each EuroSim state, tasks must be declared before use in the schedule definition file; apparently this is not the case for the reported task. Add (or move) the declaration of the task.

ES: `at line` *`nnn`*`, store` *`sss`* `has not been defined in the current state`
In each EuroSim state, stores must be declared before use in the schedule definition file; apparently this is not the case for the reported store. Add (or move) the declaration of the store.

ES: `at line` *`nnn`*`, inputconnector` *`sss`* `has not been defined in the current state`
In each EuroSim state, input connectors must be declared before use in the schedule definition file; apparently this is not the case for the reported input connector. Add (or move) the declaration of the input connector.

ES: `at line` *`nnn`*`, outputconnector` *`sss`* `has not been defined in the current state`
In each EuroSim state, output connectors must be declared before use in the schedule definition file; apparently this is not the case for the reported output connector. Add (or move) the declaration of the output connector.

ES: `at line` *`nnn`*`, this processor number falls outside the defined range of real time processors`
In the schedule definition file, a task has been allocated to a processor which is not in the range of real time processors which has been requested in the same file. Lower the processor number of the indicated task such that it falls in the range requested at the `RT_PROCESSORS` request.

E: `executer process creation failed`

Creation of a real time task executor process failed. The most probable cause of this situation is insufficient memory.

W: `cannot run executor on processor` *`p`*

Allocation of one of the real time executor processes to the specified processor failed. This message will be the result of starting `RT_SCHD` with insufficient privileges. The system will proceed in non real-time mode. When this is not intended, stop the simulation, and restart with super user authorities.

W: `too many reschedule levels for executor`

One of the internal scheduler's stacks overflows. This situation almost always occurs in combination with real time errors. Resolve the cause of the real time errors.

W: `the task activation tick of the previous cycle was still active at a new tick; this resulted in the loss of one basic cycle`

This warning is an indication that the system cannot support the requested clock frequency: the periodic part of the scheduler overruns. Reduce the clock frequency.

WS: `a preemption of the task activation tick detected; this should not have occurred`

The scheduler detected a double invocation of its periodic part, a situation which definitely should not have occurred.

W: `too few processors for specified amount of executors`

The schedule definition file requests a number of real time processors which is larger than physically available. This message is reported in combination with message `at line nnn, number of real time processors must be within the range [1...p]`. Correct the definition file by choosing a processor in the reported range, or restart with real time privileges off (no super-user authorities). This latter will result in non real-time execution mode, in which any number of 'real time' processors may be emulated.

WS: `executor table overflow`

This message indicates an overflow of one of the scheduler's internal tables. It should never occur, since the size of this table has been chosen 'sufficiently' large.

N: `execution stopped before task` *`sss`*

In debugging mode, this message reports each task which has hit a breakpoint; this task is the one which will be resumed at the next 'step' command.

WS: `taskpool was too small (extended, but this should not have occurred)`

This situation indicates that some preallocated memory in unit `Sched_TaskPool.c` is insufficient. Although it is not expected, this situation might occur in simulations with a large number of different task frequencies or task execution time bounds. The system responds to this situation by dynamically enlarging its memory resources which might theoretically result in real time errors, although the probability of this is very low. Raise an `SPR`, requesting the size of preallocated memory (`FREELIST_POOLSIZE`) in `Sched_Taskpool.c` to be raised, and continue simulating.

W: `An input event raised to connector` *`sss`* `was lost due to insufficient buffer space. Raise the capacity of this input connector in this state (currently` *`nnn`*`) and rerun the simulation`

Self explanatory.

W: `An input event raised to connector` *`sss`* `was lost due to insufficient buffer space. Raise the total capacity of the input connectors in this state (currently` *`nnn`*`) and rerun the simulation`

Self explanatory.

W: `An output event raised by connector` *`sss`* `was lost due to insufficient buffer space. Raise the total capacity of this output connector in this state (currently` *`nnn`*`) and rerun the simulation`

Self explanatory.

W: `pending state buffer overflow; state transition request ignored`

A very rapid sequence of state transition requests has caused an overflow of an internal buffer. Slow down with changing states by modifying schedule.

W: `hard real time error at uptime=`*`nnn`* `msec: periodic tasks were still active when they should have completed; basic cycle has been extended`

An overload condition has been detected, in which execution of a periodic task took longer than its allowed execution time (unless otherwise specified, this allowed time is equal to its activation period). The system responds to this situation by slowing down the real time.

Increase number of real time processors used (if possible), or decide if the effective schedule is not optimal. A schedule is not optimal if processors are unused for longer time spans[1] where this could have been avoided by a ‘smarter’ activation order of previously executed tasks. In these cases, scheduling can be influenced by processor allocation, use of task offsets and -priorities, and by adding dependencies between tasks.

W: `illegal state transition from` *`sss`* `to` *`sss`* `(ignored)`

An unallowed EuroSim state transition has been requested. It is ignored. Check the state transition diagram for legal transitions.

W: `real time mode transition refused: this machine is non real-time`

A transition of `RT_SCHD`’s mode to mode ‘real time’ has been requested in a simulation which runs with insufficient authorities, or which runs on a machine without real- time capabilities. The mode transition is ignored. Re-run with super user authorities, and use a multiprocessor platform.

W: `frequency change refused: this simulation is in real time execution mode`

A request has been given to change the clock frequency to a rate different from the rate on which the current schedule is based (200 Hz default). This request is refused in real time simulation mode. Make a mode transition to mode ‘non real-time’.

W: `frequency change refused: the requested frequency (`*`nn`* `Hz) is larger than the bound imposed by the system (`*`nn`* `Hz)`

A request has been given to change the clock frequency to a rate higher than a system-imposed bound. This has been ignored. Choose a lower rate.

W: *`itemname`* `hard real time error for` *`itemtype`* `(`*`itemdetails`*`): previous firing not completed; basic cycle has been extended`

The specified item has generated a hard real time error.

C.3 Low level errors

The errors from the previous section are scheduler run-time errors which are raised through the EuroSim message reporting mechanism. It is possible that errors occur that are not caught by this mechanism. This is usually because:

- They are raised at system initialization, when the message mechanism has not yet been initialized. These errors usually result in a text like ‘`error:` *description*’.
- They cannot be caught (e.g. bus errors, access violations). These errors usually result in a core dump.

[1] ‘Longer’ here is relative to the time granularity of the simulation, so it might apply to one or more milliseconds.

- They are on the level of code assertions, in libraries which do not 'know' the message mechanism. These errors usually result in a text like '`Assertion failed`'.

All errors of these kinds are reported through standard error, i.e. they are displayed on the console or the window in which EuroSim was started. In most cases, they indicate a problem in `RT_SCHD` and should be reported through an SPR. The second category of errors may also be caused by errors in the user code.

Appendix D

EuroSim services

This appendix describes all services and their interface description available for simulation models that want to use the EuroSim services.
These services can be used both from C as well as Fortran programs. In the latter case the function calls are all in lower or upper case (depending on your programming style). Below a short description of the available functions is given. For more information, refer to the `esim(3C)` man page.

D.1 Synopsis

D.1.1 Usage in C

```
#include <esim.h>

cc ... -L$EFOROOT/lib32 -lesServer -les
```

D.1.1.1 Real-time (shared) memory allocation

```
void *esimMalloc(size_t size)
void esimFree(void *ptr)
void *esimRealloc(void *ptr, size_t size)
void *esimCalloc(size_t nelem, size_t elsize)
char *esimStrdup(const char *str)
```

D.1.1.2 Real-time timing functions

```
double esimGetSimtime(void)
struct timespec esimGetSimtimets(void)
void esimGetSimtimeYMDHMSs(int t[7])
double esimGetWallclocktime(void)
struct timespec esimGetWallclocktimets(void)
double esimGetHighResWallclocktime(void)

int esimSetSimtime(double simtime)
int esimSetSimtimets(struct timespec simtime)
int esimSetSimtimeYMDHMSs(int t[7])
```

D.1.1.3 Real-time simulation state functions

```
esimState esimGetState(void)
int esimSetState(esimState state)
int esimSetStateTimed(esimState state, const struct timespec *t,
                      int use_simtime)
```

```
struct timespec esimGetMainCycleTime(void)
struct timespec esimGetMainCycleBoundarySimtime(void)
struct timespec esimGetMainCycleBoundaryWallclocktime(void)
```

D.1.1.4 Real-time task related functions

```
const char *esimGetTaskname(void)
double esimGetTaskrate(void)
int esimEnableTask(const char *taskname)
int esimDisableTask(const char *taskname)
int esimEntrypointFrequency(esimState state, const char *entrypoint, double freq)
int esimGetRealtime(void)
int esimSetRealtime(int on)
```

D.1.1.5 Event functions

```
int esimEventRaise(const char *eventname, const void *data, int size)
int esimEventRaiseTimed(const char *eventname, const void *data,
             int size, const struct timespec *t, int use_simtime)
int esimEventCancelTimed(const char *eventname)
int esimEventData(void *data, int *size)
int esimEventCount(const char *eventname)
```

D.1.1.6 Real-time clock functions

```
double esimGetSpeed(void)
int esimSetSpeed(double speed)
double esimGetClockfrequency(void)
int esimSetClockfrequency(double frequency)
```

D.1.1.7 Real-time recording functions

```
int esimGetRecordingState(void)
int esimSetRecordingState(int on)
```

D.1.1.8 Real-time reporting functions

```
void esimMessage(const char *format, ...)
void esimWarning(const char *format, ...)
void esimError(const char *format, ...)
void esimFatal(const char *format, ...)
void esimReport(esimSeverity lvl, const char *fmt, ...)
```

D.1.1.9 Auxiliary functions

```
const char *esimVersion(void)
void esimInstallErrorHandler(ErrorHandler userhandler)
void esimAbortNow(void)
```

D.1.2 Usage in Fortran

```
include 'esim.inc'
```

```
f77 ... -L$EFOROOT/lib32 -lesServer -les
```

The synopsis in this section uses the following variables:

```
double precision time, rate, frequency, speed
integer state, on, ok, level, counter, timespec(2), timeymd(7)
integer data(n), size, use_simtime
character*N eventname, taskname, message, version, entrypoint
```

D.1.2.1 Real-time timing functions

```
time = esimgetsimtime
time = esimgetwallclocktime
time = esimgethighreswallclocktime
call esimgetsimtimets(timespec)
call esimgetsimtimeymdhmss(timeymd)
call esimgetwallclocktimets(timespec)
ok = esimsetsimtime(time)
ok = esimsetsimtimets(timespec)
ok = esimsetsimtimeymdhmss(timeymd)
```

D.1.2.2 Real-time simulation state functions

```
state = esimgetstate
ok = esimsetstate(state)
ok = esimsetstatetimed(state, timespec, use_simtime)
call esimgetmaincycletime(timespec)
call esimgetmaincycleboundarysimtime(timespec)
call esimgetmaincycleboundarywallclocktime(timespec)
```

D.1.2.3 Real-time task related functions

```
call esimgettaskname(taskname)
rate = esimgettaskrate
ok = esimenabletask(taskname)
ok = esimdisabletask(taskname)
ok = esimentrypointfrequency(state, entrypoint, frequency)
```

D.1.2.4 Event functions

```
ok = esimeventraise(eventname, data, size)
ok = esimeventraisetimed(eventname, data, size, timespec,
                         use_simtime)
ok = esimeventdata(data, size)
counter = esimeventcount(eventname)
```

D.1.2.5 Real-time clock functions

```
on = esimgetrealtime
ok = esimsetrealtime(on)
speed = esimgetspeed
ok = esimsetspeed(speed)
```

D.1.2.6 Real-time recording functions

```
on = esimgetrecordingstate
ok = esimsetrecordingstate(on)
```

D.1.2.7 Real-time reporting functions

```
call esimmessage(message)
call esimwarning(message)
call esimerror(message)
call esimfatal(message)
call esimreport(level, message)
```

D.1.2.8 Auxiliary functions

```
call esimversion(version)
call esimabortnow()
```

D.1.3 Usage in Ada-95

```
use Esim; with Esim
```

Do not forget to check the 'Gnat Ada runtime libraries' option in the *Model:Options* window of the Model Editor (see Figure 6.4).

D.1.3.1 Real-time (shared) memory allocation

```
function EsimMalloc(Size : Size_T) return Void_Ptr
procedure EsimFree(Ptr : Void_Ptr)
function EsimRealloc(Ptr : Void_Ptr Size : Size_T) return Void_Ptr
function EsimCalloc(Nelem : Size_T Elsize : Size_T) return Void_Ptr
function EsimStrdup(Str : Chars_Ptr) return Chars_Ptr
function EsimStrdup(Str : String) return String
```

D.1.3.2 Real-time timing functions

```
function EsimGetSimtime return Long_Float
function EsimGetSimtimets return Time_Spec
procedure EsimGetSimtimeYMDHMSs(SimTime: out YMDHMSs)
function EsimSetSimtime(Simtime: Long_float) return Integer
function EsimSetSimtimets(Simtime: in Time_Spec) return Integer
function EsimSetSimtimeYMDHMSs(Simtime: in YMDHMSs) return Integer
function EsimGetWallclocktime return Long_Float
function EsimGetHighResWallclocktime return Long_Float
function EsimGetWallclocktimets return Time_Spec
```

D.1.3.3 Real-time simulation state functions

```
function EsimGetState return esimState
function EsimSetState(State : esimState) return Integer
function EsimSetState(State : esimState) return Boolean
function EsimSetStateTimed(State : EsimState;
                           T : in Time_Spec;
                           Use_Simtime : Integer) return Integer
function EsimSetStateTimed(State : EsimState;
                           T : in Time_Spec;
                           Use_Simtime : Boolean) return Boolean
function EsimGetMainCycleTime return Time_Spec
function EsimGetMainCycleBoundarySimtime return Time_Spec
function EsimGetMainCycleBoundaryWallclocktime return Time_Spec
```

D.1.3.4 Real-time task related functions

```
function EsimGetTaskname return Chars_Ptr
function EsimGetTaskname return String
function EsimGetTaskrate return Long_Float
function EsimEnableTask(Taskname : Chars_Ptr) return Integer
function EsimEnableTask(Taskname : String) return Boolean
function EsimDisableTask(Taskname : Chars_Ptr) return Integer
function EsimDisableTask(Taskname : String) return Boolean
```

D.1.3.5 Event functions

```
function EsimEventRaise(EventName : Chars_Ptr;
                        Data : Void_Ptr;
                        Size : Integer) return Integer
function EsimEventRaise(EventName: in String;
                        Data : in Void_Ptr;
                        Size : Integer) return Boolean
function EsimEventRaiseTimed(EventName : in Chars_Ptr;
                             Data : in Void_Ptr;
                             Size : Integer;
                             T : in Time_Spec;
                             Use_Simtime : Integer) return Integer
function EsimEventRaiseTimed(EventName : in String;
                             Data : in Void_Ptr;
                             Size : Integer;
                             T : in Time_Spec;
                             Use_Simtime : Boolean) return Boolean
type Integer_Ptr is access Integer;
function  EsimEventData(Data : in Void_Ptr;
                        Size : Integer_Ptr) return Integer
function EsimEventCount(EventName : String) return Integer
```

D.1.3.6 Real-time clock functions

```
function EsimGetSpeed return Long_Float
function EsimSetSpeed(Frequency : Long_Float) return Integer
function EsimSetSpeed(Frequency : Long_Float) return Boolean
function EsimGetRealtime return Integer
function EsimGetRealtime return Boolean
function EsimSetRealtime(On : Integer) return Integer
function EsimSetRealtime(On : Boolean) return Boolean
```

D.1.3.7 Real-time recording functions

```
function EsimGetRecordingState return Integer
function EsimGetRecordingState return Boolean
function EsimSetRecordingState(On : Integer) return Integer
function EsimSetRecordingState(On : Boolean) return Boolean
```

D.1.3.8 Real-time reporting functions

```
procedure EsimMessage(Warning : Chars_Ptr)
procedure EsimMessage(Warning : String)
procedure EsimWarning(Message : Chars_Ptr)
procedure EsimWarning(Message : String)
procedure EsimError(Error : Chars_Ptr)
procedure EsimError(Error : String)
procedure EsimFatal(Fatal : Chars_Ptr)
procedure EsimFatal(Fatal : String)
procedure EsimReport(S : esimSeverity Report : Chars_Ptr)
procedure EsimReport(S : esimSeverity Report : String)
```

D.1.3.9 Auxiliary functions

```
function EsimVersion return Chars_Ptr
function EsimVersion return String
procedure EsimAbortNow
```

D.2 Description of functions

When you link in the `libesim.a` library a `main()` function is already included for your convenience. It makes sure all EuroSim processes are started up.

`esimMalloc`, `esimFree`, `esimRealloc`, `esimCalloc` and `esimStrdup` are common memory allocation functions. These are the same as their `malloc`(3) counterparts in the "C" library, with the exception that the EuroSim calls are optimized for parallel/real-time usage, and checks for memory exhaustion are built-in. For the semantics and arguments and return values see `malloc`(3) for details.

`esimGetSimtime()` returns the simulation time in seconds with the precision of the basic cycle with which the simulation runs (5 ms by default). In case the simulation is driven by the external interrupt the precision is equal to that period. If the simulator has real-time errors the simulation time will be slower than the wall clock. The simulation time is set to zero (0) on arriving in initializing state.

`esimGetWallclocktime()` returns the wallclock time in seconds. The basic resolution is equal to the resolution of the high-res time described next, but is truncated to milliseconds. The wallclock time is set to zero when the first model task is scheduled, and runs real-time which means that is independent from the simulation time.

`esimGetWallclocktimets()` returns the wallclock time in a timespec structure. It replaces the obsolescent `esimGetWallclocktimeUTC()`.

`esimGetHighResWallclocktime()` returns the "same" time as `esimGetWallclocktime()` but in milliseconds and with a higher resolution. This high resolution is 21 ns on high-end platforms such as a Challenge and Onyx. On low end platforms this resolution is as good as what can be achieved by the `gettimeofday(3)` call.

`esimGetSimtimets()` returns the simulation time in a timespec structure. It replaces the obsolescent `esimGetSimtimeUTC()`.

`esimGetSimtimeYMDHMSs()` returns the simulation time in an array of 7 integers containing: year, month, day, hour, minute, second and nanoseconds.

`esimSetSimtime()` sets the requested simulation time simtime in seconds. This can only be done in the standby state. If calling esimSetSimtime in any other state is attempted or simtime is less than zero, no simulation time is set and (-1) is returned. On success zero (0) is returned.

`esimSetSimtimets()` sets the simulation time using a timespec structure. It replaces the obsolescent `esimSetSimtimeUTC()`.

`esimSetSimtimeYMDHMSs()` sets the simulation time using an array of 7 integers containing: year, month, day, hour, minute, second and nanoseconds.

`esimGetState()` returns the current simulator state. The state can be any of the following values: `esimUnconfiguredState`, `esimInitialisingState`, `esimExecutingState`, `esimStandbyState` or `esimStoppingState`.

`esimSetState()` sets the simulator state to the indicated value *state*. *state* can be any of the following values: `esimUnconfiguredState`, `esimInitialisingState`, `esimExecutingState`, `esimStandbyState` or `esimStoppingState`. If *state* is not reachable from the current state 0 is returned; on a successful state transition 1. is returned.

`esimSetStateTimed()` sets the simulator state to the indicated value *state* at the specified time *t*. The possible values of *state* are listed in the previous paragraph. If the flag *use_simtime* is set to 1 (true), the specified time is interpreted as simulation time. If the flag is set to 0 (false), the specified time is interpreted as the wallclock time. The transition time uses a struct timespec where the number of seconds is relative to January 1, 1970. On success this function returns 0, otherwise -1.

`esimGetMainCycleTime()` returns the main cycle time of the schedule. The result can be used to compute valid state transition times for use in the function `esimSetStateTimed()`.

`esimGetMainCycleBoundarySimtime()` returns the simulation time of the last state transition. This boundary time can be used to compute valid state transition times for use in the function `esimSetStateTimed()` when the value of *use_simtime* is true.

`esimGetMainCycleBoundaryWallclocktime()` returns the wallclock time of the last state transition. This boundary time can be used to compute valid state transition times for use in the function `esimSetStateTimed()` when the value of *use_simtime* is false.

`esimGetTaskname()` returns the name of your current task.

`esimGetTaskrate()` returns the frequency (in Hz) of your current task.

`esimDisableTask()` disables the task ‘taskname’ defined with the Schedule Editor. It will be skipped (not executed) by the EuroSim runtime until a call is made to `esimEnableTask`.

`esimEnableTask()` enables the task ‘taskname’ defined with the Schedule Editor. It will be executed/scheduled according to the schedule made with the Schedule Editor.

`esimEntrypointFrequency()` stores the frequency (in Hz) of the entrypoint with the name ‘entrypoint’ in the argument ‘freq’ in the state ‘state’. If the entrypoint appears multiple times in the schedule the function returns -1. If the entrypoint does not appear in the schedule in the given state, the frequency is 0.

`esimEventRaise()` raises the event *eventname* for triggering tasks defined with the Schedule Editor. User defined data can be passed in *data* and *size*. On success this function returns 0, otherwise -1.

`esimEventRaiseTimed()` raises the event *eventname* for triggering tasks defined with the schedule editor at the specified time *t*. User defined data can be passed in *data* and *size*. If the flag *use_simtime* is set to 1 (true), the specified time is interpreted as simulation time. If the flag is set to 0 (false), the specified time is interpreted as the wallclock time. The transition time uses a struct timespec where the number of seconds is relative to January 1, 1970. On success this function returns 0, otherwise -1.

`esimEventData()` gets the data passed with the event. This function can only be used in the task connected to the input connector.

`esimEventCount()` returns the number of times that event *eventname* has been raised or -1 if no such event is defined.

`esimGetRealtime()` returns the current operational state of the EuroSim real-time Scheduler. If 1 is returned, hard real-time execution is in progress, whereas a return of 0 indicates that your model is not executing in real-time mode.

`esimSetRealtime()` sets the current operational state of the EuroSim real-time Scheduler. Hard real time execution can only be set if the scheduler was launched in hard real time mode. 1 is returned on success. 0 is returned on failure.

`esimGetSpeed()` returns the current speed of EuroSim Scheduler. e.g. 1.0 means (hard or soft) real time. 0.1 means slowdown by a factor 10. -1 means as fast as possible.

`esimSetSpeed()` sets the current speed of EuroSim Scheduler. e.g. 1.0 means (hard or soft) real time. 0.1 means slowdown by a factor 10. -1 means as fast as possible. The speed can only be changed if the scheduler is running non real-time. If speed is not a feasible speed 0 is returned; on a successful setting of the speed 1 is returned.

`esimGetRecordingState()` returns the current state of the EuroSim real-time data Recorder. If *true* is returned, data is logged to disk, whereas a return of *false* indicates that recording is switched off.

`esimSetRecordingState()` sets the state of the Recorder to *on*. If *on* is *true* data will subsequently be written to disk, if *on* is *false* data recording will be suspended. Return value is either *true* or *false*, depending on success or not.
The functions `esimReport`, `esimMessage`, `esimWarning`, `esimError` and `esimFatal` can be used to send messages from the EuroSim model code to the test-conductor interface. The `esimReport` function allows the caller to specify the severity of the message. The other functions have implicit severities. The possible *severity* levels are:

- esimSevMessage for comment or verbose information
- esimSevWarning for warnings
- esimSevError for errors
- esimSevFatal for non-recoverable errors

In the C interface routines the message consists of a format string *format* and its optional arguments. (see `printf`(3)). In the Fortran interface routines the message consists of a single string argument *message*.

`esimVersion()` returns a string indicating the current version of EuroSim that you are running.

`esimInstallErrorHandler()` installs a user-defined error handler callback of the form:

```
void userhandler(esimErrorScope scope,
                 const char *objectid)
```

This callback function is called when an error occurs that may need intervention in user code.
Passing a NULL pointer will de-install the user error handler. No stack of user error handlers is maintained. This means that the last call to `esimInstallErrorHandler` defines which handler will be called.
The possible values for *scope* are:

- esimDeadlineError when the user defined error handler is called with this scope then the *objectid* is the name of a task in the simulator schedule that has exceeded its deadline by a factor of ten. This allows a model developer to take action (f.i. force a core dump) when part of a model is ill-behaved (never ending loops or simply a calculation that takes too long and causes real-time errors). If no error handler is installed, the default action is to disable the offending task and enter the stand-by state. Note that deadline checking is only performed when the simulator is running in real-time mode.

`esimAbortNow()` immediately starts termination and cleanup of the simulator. This is useful when an error condition is found (f.i. at initialisation time) and no more entrypoints should be scheduled for execution.

D.3 Simulator Options

The following are the command line options that can be passed to the `main()` function of the simulator:

`-h`	Show on-line help and exit.
`-v`	Enable verbose printing of currently running simulator.
`-u` *uid*	(Numeric) uid for file ownership of recordings etc.
`-g` *gid*	(Numeric) gid for file ownership of recordings etc.
`-c` *number*	Connection number offset for the simulator process. Needed if more simulators are to be run on one host.
`-x` *simfile.sim*	Simulation definition to initially load.
`-m` *scenario.mdl*	Scenario to initially load.
`-e` *exportsfile.export*	Exports file for ExtSimAccess.
`-i` *initialcond.init*	Initial condition file to load.
`-d` *datadict.dict*	Data dictionary to load.
`-s` *schedule.sched*	Schedule file to load the scheduler with.
`-f` *number*	Frequency to run the asynchronous processes with. The default for the asynchronous frequency is 2 Hz.
`-R` *directory*	Directory to write recording files to. Defaults to the directory where the *simfile.sim* file came from.
`-l` *number*	Period (with respect to asynchronous frequency) for datalogger. Every *number*'th cycle data values will be delivered to (interested) clients (e.g. a simulation controller with a datamonitor). Defaults to 1.
`-r` *number*	When *number* is 0, real-time mode is off, when it is 1 it is on.
`-D` *flags*	Debugging flags. Only available when EuroSim libraries were compiled with `DEBUG` defined.
`-M` *modelfile.model*	The name of the model file used to create the simulator.
`-E`	Don't use the daemon for services (CPU allocation).
`-I`	Do not go to initializing state automatically.
`-S`	Stand-alone mode (do not wait for client to connect). You may want to use this flag in combination with the `-E` flag. Useful for debugging from the command line with i.e. `gdb`.

Note that under normal circumstances the above options will be passed to the simulator by the EuroSim daemon.
Example of a debugging session, running the simulator from the command line using gdb. Note that you must have root privileges.

```
$ gdb mysimulator.Linux/mysimulator.exe

(gdb) run -c 1 -x mysimulator.sim -s mysimulator.sched
        -d mysimulator.Linux/mysimulator.dict
        -M mysimulator.model -r 1 -user 18157 -g 100
        -R result_dir -v -f 10 -E -S
```

Appendix E

Mission Definition Language

The Mission Definition Language MDL is a simple yet versatile language for simulation scripting. It allows users to write simulator control scripts in a “C-type”, or—alternatively—in a limited “free-text” language. The language has all the facilities one can expect of a programming language, including if-statements, for-loops, global and local variables. Besides that, the user has full access to the variables in the EuroSim data dictionary. Direct simulation control commands can also be used in the language.
This appendix first starts with a primer in MDL, followed by a number of sections providing detailed information on the various language elements. A description of the built-in functions and a concise formal definition of the MDL language can be found in the last two sections of this appendix.
Note that the majority of MDL scripts in EuroSim will/can be made via the GUIs, for which the user doesn’t need to know much about the MDL language. So this appendix is primarily intended for EuroSim users who want to do ‘advanced’ things, not supported via the predefined GUIs. Throughout this section, it is assumed that the reader has programming experience.

E.1 MDL primer

An MDL script (or “scenario”) is normally created with EuroSim’s Simulation Controller and interpreted during simulation by EuroSim’s Action Manager (ACTION_MGR). An MDL script contains (amongst other things) a collection of *actions*. An MDL action consists of four parts:

1. Action name.
2. Action attributes (optional).
3. Action body.
4. Action condition (optional).

Each action in the MDL script is represented by an icon on the Simulation Controller’s tree or icon view. The four parts of each action can be edited via the Simulation Controller (Section 12.13).
A simple example which prints a message 10 seconds into the simulation:

```
#
# action name and attributes
action "Primer" ["description",bitmap="script_stub",show+active+Executing
    , 50 50, 1]
#
# action body
{
  print "Hello at t=10"
}
#
# action condition
when (time() == 10)
```

The action attributes are used to:

- Give a description of the action.
- Manipulate the appearance of the action on the Simulation Controller tree or icon view.
- Set the *initial*[1] status of the action.

The action status can either be `active` or `nonactive`. Furthermore, one can specify in which of the four simulation states the action has to be evaluated when active: `Initializing`, `Executing`, `StandBy` or `Stopping`.
EuroSim maintains for each of these states a list of active actions. The action conditions of these actions are checked each time the `ACTION_MGR` is activated (in that state and normally at the end of each simulation step[2]). The action body is executed by EuroSim when the action condition evaluates true. When the action has no condition part, this never happens; these actions can only be activated manually (by double clicking the action icon on the Simulation Controller scenario tab page).
The MDL script is executed in the real-time part of EuroSim. In order to safeguard the real-time execution of a EuroSim simulator, error conditions within MDL actions are handled in the following way:

1. The execution of the action causing the error condition is suspended.
2. An error message of this event is reported to the Test Controller and the journal log.
3. The specific action is deactivated so the action will not be executed again.
4. The execution of remaining actions in the MDL script is resumed.

Run time error conditions include:

- MDL or data dictionary array bound overflows.
- Errors in MDL math functions or expressions (e.g. `sqrt(i)` with $i<0$).
- Errors in action condition frequency specification (e.g. frequency higher than the `ACTION_MGR` frequency).
- Trying to read stimuli from nonexisting or exhausted stimuli files.
- Observers (which have “read only” access) trying to change the data dictionary variables from actions, apply stimuli or raise events.
- MDL scripts accessing undefined (external) MDL variables or functions.
- MDL scripts trying to execute an undefined action.

An MDL action body consists of statements separated by newlines or by a semicolon. The latter may—however—only be used to separate multiple statements on a single line. MDL is case sensitive. Everything following a ‘#’ sign until end-of-line is considered comment.
MDL is a powerful languages, but remember that it is an interpreted language, running in the real-time part of the simulator. Hence keep your scripts as simple and small as possible. Don’t write large loops and keep computation to a minimum. If you have to do serious programming and/or computation, consider adding an extra sub-model and associated tasks to you model.

[1]Initial, as this can change during the simulation.
[2]See Section 11.3.5 for scheduling of the `ACTION_MGR`.

E.2 MDL constants, types, variables, operators and expressions

Variable names are made up of letters, underscores and digits. Upper and lower case letters are distinct. MDL has four basic variable types:

- `int` representing an integer value.
- `float` representing a floating point value[3].
- `string` representing a character string.
- `datetime` representing a time value.

Variables which are explicitly declared as one of the above are called 'static' variables. Static variables are, in the absence of an initializer, always initialize at zero or the empty string. Variables need not be declared in MDL. Undeclared variables are created automatically the first time they are used as a left hand value in an assignment. These variables are called 'automatic' variables.
The scope of variables is that of the enclosing action body (or function; see below). By prepending the action or function name, the static variables from other actions and functions can be accessed. Static variables retain their values in between different action or function invocations. Automatic variables are recreated each time their scope is entered and disappear when that scope is left.
Automatic type conversions are applied when needed between all the basic types.
Constants can be given either in decimal, octal or hexadecimal form, as in 'C'. Constants are of type int, except when the constant contains a decimal dot or is given in scientific notation (e.g. 3e-9), in which case they're of type float. A string constant consists of a number of characters between double quotes.
Some examples with MDL variables and constants:

```
action "action1"
{
  int a_variable      # a static variable of type int
  b_variable = "100"  # an automatic variable of type string
  a_variable = b_variable # type conversion from string to int
}

action "action2"
{
   string a_variable = "hello" + " world" # an initialised static variable
}

action "two_externals"
{
  float f = action1:a_variable
  print f             # prints: "100.0000"
  print action1:a_variable # prints: "100"
  print action2:a_variable # prints: "hello world"
}

action "showtime"
{
  # NB: No UTC selected
  datetime t = 5.900
  print "time = ",t + 15.2 # prints: "time = 21.1000"
}

action "showtimeUTC"
{
  # NB: UTC selection in model options
```

[3]`int` and `float` are implemented as C doubles; check the documentation of your platform to see the valid range for that type.

```
  datetime t = 2001-02-24 16:10:05.900
  print "time = ",t + 15.2 # prints: "time = 2001-02-24 16:10:21.1000"
}
```

Arrays of basic types can be constructed using square brackets. Arrays must have fixed dimensions and type (no automatic arrays). Assignments are between basic types only.

```
action "sum"
{
  int a[10]
  for (i = 0; i < 10; i = i + 1) a[i] = i

  # compute sum of array
  i = 0; sum = 0
  while (i < 10) {
    sum = sum + a[i]; i = i + 1
  }
}
```

MDL has all the usual (C) operators, except for the address operator, which doesn't exist in MDL. Exponentiation is written as 3^4. In addition, the equivalent English words can be used as operators, e.g. `and`, `or`, `not`, `less_equal`, `greater_equal`, `equals`, `not_equals`, `less_than`, `greater_than`, `minus`, `plus`, `times`, `pow`.

E.3 Control Flow

MDL statements within an action body are executed in order from top to bottom, except as modified by control flow statements. MDL has the usual (C) keywords for control flow: `break`, `continue`, `do`, `else`, `for`, `if`, `while`, `return`. There's no switch-construct (yet), although the words 'switch' and 'case' are reserved words[4]. A conditional block (sequence of statements) may be delimited by either curly braces '{}' or by the keywords `begin` and `end`. The action body may be delimited by the keywords `action_begin` and `action_end`. These latter two keywords may thus not be nested, and help (when used) to find nesting problems, which are then confined to a single action in the MDL script. Below two examples are given, one in C-like syntax, and one in the alternative, free-style syntax.

```
action "looptest2"
{
  j = 0
  N = 100

  print ""
  print "# forloop test2, expect loopcount=", N

  for (i = 1; i < 10 * N; i = i + 1) {
    j = j + 1
    if (i == N) break;
  }
  print "loopcount=", j
}

# free-style syntax
action "looptest5"
action_begin
  N = 3000
  k = 0

  print ""
  print "# forloop test5, expect loopcount=", N
```

[4]See Section E.6 for a complete list of reserved, but unused words.

```
  for i is 1 to N/10 loop begin
    for j is 1 to 10 loop begin
      k is k plus 1
    end
  end

  print "loopcount=", k
action_end
```

E.4 Functions

MDL has an extensive set of built-in functions for simulation support: see Section E.6. It also supports user defined functions. Functions return simple values and can be used freely in MDL expressions.
User functions can be defined and used within the action body. Function arguments and return values must be basic types and behave like automatic variables. Within the function body the complete MDL syntax can be used (e.g. to define local variables or other functions).
The type of the function arguments and the type returned by the function may vary from invocation to invocation, as is shown in next example.

```
action "my_action"
{
  int i
  float x
  error = 0

  function sqr(n)
  {
    return n * n
  }

  for (i = 0; i < 5; i = i + 1) {
    # sqr with int
    if (sqr(i) != i * i) error = error + 1
  }

  for (x = 0.0; x < 5.0; x = x + 1.0) {
    # sqr with float
    if (sqr(x) != x * x) error = error + 1
  }

  if (!error) print "function test OK"
  else print "Error !!!"
}
```

The scope of the function name is that of the enclosing action. As with variables, one can use a function defined in another action, by pre-pending that action's name and a colon (':') to the function's name.

```
# simple external function call
action "object"
{
  float velocity = 10.0 # static variable
  function speedup() { velocity = velocity * 2.0; }
  function slowdown() { velocity = velocity * 0.5; }
  function current() { return velocity; }
}

action "accel"
{
  object:speedup()
  print "speed=", object:current() # prints: "speed=20.0"
}
```

Warning: because all MDL variables have static storage, recursive function calls may have unexpected results.

E.5 Input/Output and Simulator Control

In MDL, input and output can be done in two ways, each having a particular purpose:

1. Via variables in the simulation model.
2. Via specific built-in commands.

An example of the latter is the `print` command, already shown in many of the previous examples. It prints the given expression on the Simulation Controller's message pane and in the simulation log.

MDL provides access to variables in the simulation model via the model's data dictionary. Array elements are selected using square (C) or round (Fortran) brackets. More dimensional array indexing follows the conventions of the sub-model language. Members of user defined type variables in C sub-models are selected using a dot:

```
action "position"
{
  # print three elements of an array in a Fortran style loop
  N = 3
  for i is 1 to N loop begin
    print "position(", i, "): ", :source.f:position(i)
  end
}

action "clear"
{
  # clear all elements of an 2-dim. array in a C style loop
  for (i = 0; i < 10; i = i + 1)
    for (j = 0; j < 10; j = j + 1)
      :source.c:matrix[i][j] = 0

  # clear a member of a C struct
  :source.c:vector.xcoord = 0.0
}
```

A combination of both mechanisms is used to stimulate and record certain data dictionary variables with the `stimulate` or `record` built-in commands.

```
action "register three"
{
  int n
  float x

  function f()
  {
    x += 1.0
    return x
  }

  n++;
  record "file1" n, f(), :A:B:C:source1.c:work1:local1
  record "file3" :A:E:C:source2.c:work4:localUdt
  record "file2" :A:E:C:source2.c:work4:localUdt[0].count
}
when (freq(100))
```

Note that also MDL variables can be recorded; this can be used e.g. for recording a derived variable (derived from one or more data dictionary variables).
From within MDL, the user has full control over the simulator by means of functions like `go`, `freeze`, `stop`, etc. (see Table E.5) Also, from one action, one can activate other actions but also tasks and entry points within the model.

E.6 MDL Built-in functions and commands

MDL has built-in functions and commands for the following applications:

- Mathematical functions (see Table E.2).
- Signal processing functions (see Table E.3).
- Auxiliary functions (see Table E.4).
- Input, output and control commands (see Table E.5).

Functions return a value, whereas commands do not. Functions can be used in expressions. The MDL built-in functions all take numerical (or no) arguments. Required arguments are indicated as follows:

`func()`	This function takes no argument.
`func(x)`	This function takes one argument.
`func(x, ...)`	This function takes one or more arguments.

Arguments may be functions themselves. Non-numerical arguments are automatically converted to numerical.

Function	Description
`atan``(x)`	Compute arc tangent of `x` and return it. Return value will be between $-\pi/2$ and $\pi/2$.
`cos``(x)`	Compute cosine of `x` and return it. `x` is in radians.
`exp``(x)`	Compute the `x`'th power of `e` and return it. `e` is the base of natural logarithms.
`fabs``(x)`	Compute the absolute value of `x` and return it.
`log``(x)`	Compute the natural logarithm of `x` and return it. If `x` is less than or equal to 0, a run time error results.
`sin``(x)`	Compute the sine of `x` and return it. `x` is in radians.
`sqrt``(x)`	Compute the square root of `x` and return it. If `x` is less than 0, a run time error results.
`tan``(x)`	Compute the tangent of `x` and return it. `x` is in radians.
`acos``(x)`	Compute the arc cosine of `x` and return it. Return value will be between 0 and π. If `x` is not between -1 and 1, a run time error results.
`asin``(x)`	Compute the arc sine of `x` and return it. Return value will be between $-\pi/2$ and $\pi/2$. If `x` is not between -1 and 1, a run time error results.
`ceil``(x)`	Rounds up `x` to the next highest integer and return it.
`cosh``(x)`	Compute the hyperbolic cosine of `x` and return it.
`floor``(x)`	Rounds down `x` to the next lowest integer and return it.
`log10``(x)`	Compute the (base 10) logarithm of `x` and return it. If `x` is less or equal than 0, a run time error results.
`sinh``(x)`	Compute the hyperbolic sine of `x` and return it.

Table E.2: Mathematical functions.

Function	**Description**
`tanh``(x)`	Compute the hyperbolic tangent of `x` and return it.

Table E.2: Mathematical functions.

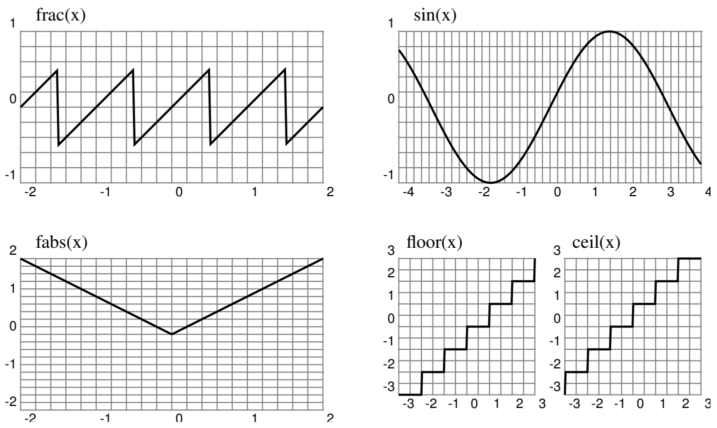


Figure E.1: Some of MDL's mathematical functions.

Function	**Description**
`doublet``(x)`	Compute the doublet of `x` and return it. If `x` is between 0 and 1 return 1, if `x` is between 0 and -1 return -1, else return 0.
`ramp``(x)`	Compute the ramp of `x` and return it. If `x` is less than zero return zero, if `x` is greater than 1 return 1, else return `x`.
`jigsaw``(x)`	Compute the jigsaw of `x` and return it. If `x` is less than 0 return 0, if `x` is greater than 1 return 0, else return `x`.
`step``(x)`	Compute the step of `x` and return it. If `x` is less than 0 return 0, if `x` is greater than 0 return 1.
`frac``(x)`	Compute the frac of `x` and return it. Frac is the remainder of `x` from its nearest integer value.

Table E.3: Signal processing functions.

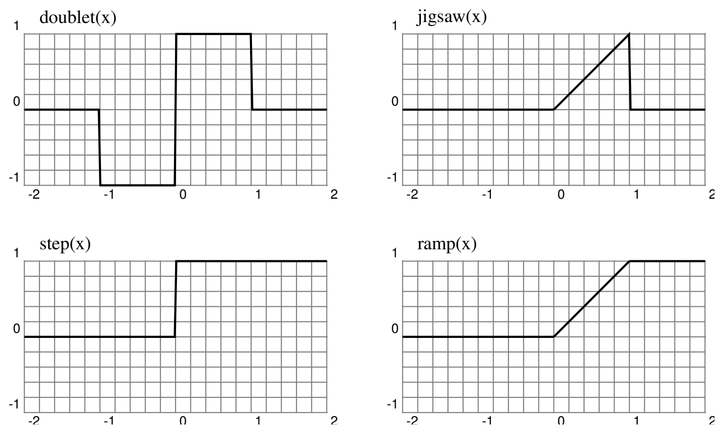


Figure E.2: Some of MDL's signal processing functions.

By combining (or modulating) the various functions in expressions, many types of signals and *if-type* functions can be constructed. For example:

- `step(x+1)-step(x-1)` or `doublet(x)*doublet(x)` results in the box function which is only 1 in the range [-1, 1], and 0 everywhere else.
- `x*step(-x)+x*x*step(x)` results in a line for `x` less than zero and a parabola for `x` greater than zero.

Function	**Description**
`catch``(x)`	Reserved for future use.
`changed``(x)`	Return 1 if `x` has changed with respect to the previous invocation, else return 0. Typically used in the condition part of an action in combination with data dictionary variables: `freq(100) & changed(:model:var)`
`duration``()`	Return the elapsed simulation time (in seconds) that the action has been continuously (i.e. at each activation of the `ACTION_MGR`) executed. Elapsed time is reset to zero when the action is not executed. This function can be used to have an action run for a certain period of time.
`format``(x,...)`	Return formatted string, using `printf` like format specification. E.g. `str = format("Hex value=%4x", :model:var)`
`freq``(x)`	Use this function to have an action executed at a given frequency. It returns 1 if desired frequency `x` (in Hertz) is met by internal basic frequency, else `freq` returns 0. The basic frequency is the frequency with which `ACTION_MGR` is scheduled. Depending on the scheduling table used, this frequency may differ from the scheduler basic frequency. If the basic frequency is not an exact multiple of the desired frequency `x` the desired frequency will be approximated in the long run. When parsing an action with a `freq` function, the `ACTION_MGR` will issue a warning if this is the case (provided `x` is a constant).
`getenv``(x)`	Return the string value of shell environment variable `x`.
`realtime``()`	Return the current real-time mode of the simulator.

Table E.4: Auxiliary functions.

Function	Description
simstate()	Return current simulator state as string value, e.g. `"standby"`. Can be used by actions which can execute in different simulator states in expressions like `if (simstate() = "executing") count = count + 1`
time()	Return the current simulation time in seconds.
wallclock()	Return the current wallclock time in seconds.
main_cycle()	Return the main cycle time of the schedule in seconds.
simtime_boundary()	Return simulation time of last state transition.
wallclock_boundary()	Return wallclock time of last state transition.

Table E.4: Auxiliary functions.

The last table explains the MDL commands. The commands take numerical or string arguments. Contrary to functions, the command arguments are not to be given between parenthesis and commands do not return a value. Hence they cannot be used in expressions.

Command	Description
abort	request abort of the simulator

activate *action* | *task*	activate an action (i.e. make its state active) or enable a task. Actions and tasks must be specified as strings:
	`activate "Inject Error"`
	`activate "task:Thruster"`

deactivate *action* | *task*	deactivate an action or disable a task. Actions and tasks must be specified as strings:
	`deactivate "Inject error"`
	`deactivate "task:Thruster"`

exec *action* | *entrypoint* | *task*	execute action or model entrypoint or model task from within another action. Action, entrypoints and tasks must be specified as strings:
	`exec "Trigger action"`
	`exec "entry:do_step"`
	`exec "task:my_task"`

health	check internal diagnostics and report it to the journal file

mark [*expression*]	Produce a mark in the message pane and journal file. When expression is omitted, the mark looks like:
	`MARK-n,`
	with n being a sequence numberWhen expression is given, the mark looks like:
	`COMMENT-n comment,`
	with comment being the value of expression converted to string.

Table E.5: Input/Output and Control commands (do not return values)

Command	**Description**
`monitor` `[`*`options`*`]` *`dictlist`*	Please note that this command is obsolescent. Pop-up a monitor on the “Script Monitors” tab pane. This command can be used to start monitoring of a (set of) variable(s) when a certain condition during simulation is met. Information on the variables is derived from the data dictionary. The *options* argument is a single string containing a comma separated list of options. Valid options are: type alfa|time|xy: type of monitor point cross|line|both: line style of monitor xsize *number*: xsize of monitor window ysize *number*: ysize of monitor window xmin *number*: minimum x value of monitor plot range xmax *number*: maximum x value of monitor plot range ymin *number*: minimum y value of monitor plot range ymax *number*: maximum y value of monitor plot range Example: `monitor "type=time, point=cross, xsize=1, ysize=2, xmin=3.0, xmax=4, ymin=5, ymax=6" :A:B:C:source1.c:work1:local1`
`pause` `(or` **`freeze`**`)`	request change simulator state from ‘executing’ to ‘standby’.
`print` *`expression_list`*[5]	Evaluate the expressions in the *expression_list* and print them on the message pane and journal file.
`raise` *`event`*	raise an input event as defined in the EuroSim schedule, e.g.: `raise "HARDWARE_FAILURE"`
`record` `[`*`per_switch`*`]` `[`*`filename`*`]` *`dictlist`*[6] `(or` **`registrate`**`,` **`datalog`**`)`	Record one sample of a given set of variables to an optionally named file. The simulation time is recorded implicitly and need not be specified. The optional *per_switch* argument specifies the time (in seconds or hours) in case a recorder file should periodically switch. The *filename* argument is optional. It can be used for “named” recording. If *filename* is not specified, the action’s name suffixed by `.rec` will be used as file name. In case of a periodic switch the *filename* becomes *filename-00n* (with switch counter *n*).
`reinit` `["soft" | "hard"]` *`filename`* `(or` **`initialise`**`,` **`init`**`)`	Reload the data dictionary with the values from a snapshot file. If the “hard” option is given the simulation time will be set to the value defined in the snapshot file. The “soft” option is the default. When this option is used (or no option) the simulation time in the snapshot file is ignored. After the loading of the file has finished the scheduler event `SNAPSHOT_END` is raised so that a task can be triggered to use the values to reinitialize external hardware for instance.

Table E.5: Input/Output and Control commands (do not return values)

[5] An *expression_list* is a comma-separated list of expressions.
[6] A *dictlist* is a comma-separated list of data dictionary variables.

Command	**Description**
`run` `(or` **`go`**`)`	Request change simulator state from 'standby' to 'executing'
`schedspeed` *`expression`*	Set the scheduler speed to result of expression. `schedspeed("AFAP")` sets the scheduler in 'as fast as possible' mode. This function only has effect if the scheduler is in non-realtime mode.
`set_realtime` *`expression`*	Change the real-time mode to result of expression.
`set_time` *`expression`*	Change the simulation time to result of expression.
`snapshot` `[`*`filename`*`]`	Make a snapshot of the current data dictionary and save it to a file. Default file name is snapshot-n.snap, n=0, 1, 2, ...
`stimuli` `["soft" | "hard" | "cyclic"]` *`filename dictlist`*	Stimulate the specified set of data dictionary variables with the next record of values contained in *filename*. If the "hard" option is given, the next record in the stimuli file will be applied when the given timestamp (value in first column in the stimuli file) matches the simulation time. In the default case "soft" the timestamps are ignored. With the "cyclic" option the stimulation is applied periodically, ignoring the timestamps.
`stop`	request change simulator state to 'stopping'

Table E.5: Input/Output and Control commands (do not return values)

E.7 MDL syntax

The syntax below is specified in a Backus-Naur Form.
':' indicates the start of the definition of the item listed before the colon. '|' indicates an alternative and ';' terminates the definition. So A: B|C|D; means that 'A' can be 'B', 'C' or 'D'.
`Bold` words are literal strings.

`string` is a placeholder for an actual string, i.e. a sequence of characters, delimited by double quotes.
Example: `"this is a string"`

`identifier` is a placeholder for an actual identifier of a variable or function. Identifiers consist of a sequence of letters, digits and the underscore and dollar character.
Examples: `var1`, `_var2` and `block$var3`

`external-identifier` is a placeholder for an actual identifier of a variable or function coming from another MDL action. It consists of the name of an action followed by an identifier of a variable or function separated by a colon. The name of an action may contain spaces and therefore it is possible to enclose the name of the action in double quotes. If there are no spaces in the name of the action the double quotes are not needed.
Examples: `action1:var1` and `"action two":var2`

`dictpath` is a placeholder for a data dictionary path name. A data dictionary path consists of a list of orgnodes followed by an identifier separated by colons.
Example: `:system-A:subsystem-B:source.c:variable_d`

{**`Decimal`**} is a decimal number.

{**`Octadecimal`**} is an octal number. It starts with a 0 and consists of one or more numbers in the range 0-7.

{**`Hexadecimal`**} is a hexadecimal number. It starts with 0x and consists of one or more numbers in the range 0-9 and letters in the range A-F or a-f.

{**FloatingPoint**} is a floating point number. It can have a decimal point and/or an exponent.

{**Time**} is a time specification. It has the following format: YYYY-MM-DD hh:mm:ss optionally followed with a decimal point followed by fractions of a second. YYYY is the year in four decimal digits. MM is the month of the year in the range of 1 to 12. DD is the day of the month in the range of 1 to 31. hh is the hour of the day in the range of 0 to 23. mm is the minute of the hour in the range of 0 to 59. ss are the seconds of the minute in the range of 0 to 59. You can specify sub-second precisions by adding a fraction to the seconds.
Example: `2003-06-05 10:11:12.131415`

```
#grammar:
MDL
         /* MDL action scripts */
         : MDLscript tEOF
         | MDLfuncs Cont MDLscript tEOF
         | tEOF
         ;

MDLscript
         : Action
         | MDLscript Action
         ;

MDLfuncs
         : FunctionDeclaration
         | MDLfuncs Term FunctionDeclaration
         ;

Action
         : action string
         /* CONTINUED */
         Attributes Cont
         /* CONTINUED */
         ActionBody
         Cont
         /* CONTINUED */
         ActionCondition
         ;

ActionBody
         : CompoundStatement
         | action_begin Cont StatementList Cont action_end
         | action_begin Cont action_end
         | { Cont }
         | begin Cont end
         | action_begin Cont StatementList tEOF
         | { Cont StatementList tEOF
         | begin Cont StatementList tEOF
         ;

Attributes
         : /* no attributes */
         | [ AttributeList ]
         ;

ActionCondition
         : /* no condition */
         | When ( Cont PossibleCondition ) Term
         ;

When
         : when
         ;

PossibleCondition
```

```
         : /* nothing */
         | Condition Cont
         ;

Condition
         : Expr
         ;

         /*
         * ActionAttribute are used to manipulate:
         * - appearance of Action in Action Sheet (GUI)
         * - initial status of action, when condition is specified
         *
         */
AttributeList
         : ActionAttribute
         | AttributeList , ActionAttribute
         ;

ActionAttribute
         : ActionStateAttribute           /* state attributes */
         | PixelCoord PixelCoord          /* x y position on action Sheet */
         | {Decimal}                      /* icon # on action Sheet */
         | {Octadecimal}                  /* icon # on action Sheet */
         | {Hexadecimal}                  /* icon # on action Sheet */
         | bitmap is string               /* bitmap */
         | bitmap = string                /* bitmap */
         | index is {Decimal}             /* index */
         | index = {Octadecimal}          /* index */
         | index is {Hexadecimal}         /* index */
         | folder is string               /* folder */
         | folder = string                /* folder */
         | actionmgr is {Decimal}
         | actionmgr = {Octadecimal}
         | actionmgr is {Hexadecimal}
         /* action mgr nr */
         | type is string
         | type = string
         | string                         /* description field */
         ;

ActionStateAttribute
         : identifier
         | ActionStateAttribute || identifier
         | ActionStateAttribute or identifier
         | ActionStateAttribute + identifier
         | ActionStateAttribute plus identifier
         ;

CompoundStatement
         : { Cont StatementList Cont }
         | begin Cont StatementList Cont end
         ;

StatementList
         : Statement
         | StatementList Statement
         ;

Statement
         : DeclarationList
         | for ( Assignment ; Condition ; Assignment
         ) Cont Statement
         | for Assignment to Expr loop Cont Statement
         | while ( Condition ) Cont Statement
         | do Statement while ( Condition ) Term
         | do CompoundStatement while ( Condition ) Term
         | continue Term
         | break Term
         | return Term
         | return Expr Term
         | Assignment Term
         | BuiltInCommand Term
         | FunctionCall Term
         | ; Term
         | IfStatement
         | CompoundStatement Term
         ;

IfStatement
         : if ( Condition ) Cont ThenStatement ElseStatement
         | if ( Condition ) Cont ThenStatement
         ;

ThenStatement
         : Statement
         ;

ElseStatement
         : else Cont Statement
         ;

Assignment
         : Lvalue is Cont Expr
         | Lvalue = Cont Expr
         | set Lvalue to Expr
         | set Lvalue Expr
         | ComplexAssignment
         ;

ComplexAssignment
         : Lvalue += Expr
         | Lvalue -= Expr
         | Lvalue *= Expr
         | Lvalue /= Expr
         | Lvalue %= Expr
         ;

Lvalue
         : Variable
         ;

Expr
         : MdlExpr
         | Variable
         ;

Argument
         : MdlExpr
         | Variable
         ;

MdlExpr
         : Constant
         | FunctionCall
         | ( Expr )
         | Expr + Expr
         | Expr plus Expr
         | Expr - Expr
         | Expr minus Expr
         | Expr / Expr
         | Expr * Expr
```

```
          | Expr times Expr
          | Expr % Expr
          | - Expr
          | minus Expr
          | Expr ^ Expr
          | Expr pow Expr
          /* conditions */
          | Expr == Expr
          | Expr equals Expr
          | Expr != Expr
          | Expr not_equals Expr
          | Expr >= Expr
          | Expr greater_equal Expr
          | Expr <= Expr
          | Expr less_equal Expr
          | Expr > Expr
          | Expr greater_than Expr
          | Expr < Expr
          | Expr less_than Expr
          | Expr && Expr
          | Expr and Expr
          | Expr || Expr
          | Expr or Expr
          | Expr & Expr
          | Expr | Expr
          | Expr << Expr
          | Expr >> Expr
          | !  Expr
          | not Expr
          ;

FunctionCall
          : BuiltInFunction
          | UserFunction
          | ExternalFunction
          ;

UserFunction
          : identifier ( )
          | identifier ( ExprList )
          ;

ExternalFunction
          : external-identifier ( )
          | external-identifier ( ExprList )
          ;

BuiltInFunction
          : wallclock_boundary ( )
          | realtime ( )
          | time ( )
          | duration ( )
          | simstate ( )
          | wallclock ( )
          | main_cycle ( )
          | simtime_boundary ( )
          | getenv ( Expr )
          | atan ( Expr )
          | cos ( Expr )
          | exp ( Expr )
          | fabs ( Expr )
          | log ( Expr )
          | sin ( Expr )
          | sqrt ( Expr )
          | tan ( Expr )
          | acos ( Expr )
```

```
         | asin ( Expr )
         | ceil ( Expr )
         | cosh ( Expr )
         | floor ( Expr )
         | log10 ( Expr )
         | sinh ( Expr )
         | tanh ( Expr )
         | doublet ( Expr )
         | ramp ( Expr )
         | jigsaw ( Expr )
         | step ( Expr )
         | frac ( Expr )
         | freq ( Expr )
         | catch ( Expr )
         | eventcount ( Expr )
         | format ( ExprList )
         | changed ( Expr )
         ;

BuiltInCommand
         : activate Expr
         | deactivate Expr
         | exec Expr
         | raise Expr
         | set_time Expr
         | set_realtime Expr
         | schedspeed Expr
         | print ExprList
         | monitor string ExprList        /* string contains options (obsolescent) */
         | monitor ExprList               /* no display options (obsolescent) */
         | stimuli string ArgumentList    /* first arg is file name */
         | stimulate string ArgumentList  /* first arg is file name */
         | stimuli string string ArgumentList
         | stimulate string string ArgumentList
         | record string string ArgumentList
         | datalog string string ArgumentList
         | registrate string string ArgumentList
         | record string ArgumentList
         | datalog string ArgumentList
         | registrate string ArgumentList
         | record ArgumentList
         | datalog ArgumentList
         | registrate ArgumentList
         | initialise Expr
         | reinit Expr
         | init Expr
         | initialise string Expr
         | reinit string Expr
         | init string Expr
         | AtomicAction
         ;

AtomicAction
         : run
         | go
         | pause
         | freeze
         | stop
         | abort
         | snapshot Expr
         | snapshot
         | mark Expr
         | mark
         | health
         ;
```

```
ExprList /* list of simple expressions */
         : Expr
         | ExprList , Cont Expr
         ;

ArgumentList
         /* list of generic (may contain complex types expressions */
         : Argument
         | ArgumentList , Cont Argument
         ;

IdentList
         : identifier
         | IdentList , identifier
         ;

Constant
         : string
         | {Decimal}
         | {Octadecimal}
         | {Hexadecimal}
         | {FloatingPoint}
         | {Time}
         | zero
         | off
         | on
         ;

PixelCoord
         : {Decimal}
         | {Octadecimal}
         | {Hexadecimal}
         | + {Decimal}
         | plus {Octadecimal}
         | + {Hexadecimal}
         | - {Decimal}
         | minus {Octadecimal}
         | - {Hexadecimal}
         ;

Variable
         : identifier
         | identifier ArraySelector
         | ExternalVar
         | DictVar
         ;

ExternalVar
         : external-identifier
         | external-identifier ArraySelector
         ;

DictVar
         : DictPath
         | DictPath DictSelectorList      /* ctype selectors */
         | DictPath ( ArgumentList )      /* fortran array */
         ;

DictPath
         : dictpath
         ;

DictSelectorList
         : DictSelector
         | DictSelectorList DictSelector
         ;
```

```
DictSelector
         : RecSelector
         | ArraySelector
         ;

RecSelector
         : .identifier
         | RecSelector .identifier
         ;

DeclarationList
         : Type identifier Term
         | Type identifier is Expr Term
         | Type identifier = Expr Term
         | Type identifier ArraySelector Term
         | FunctionDeclaration Term
         ;

FunctionDeclaration
         : function identifier ( IdentList ) Cont
         CompoundStatement
         | function identifier ( ) Cont
         CompoundStatement
         ;

ArraySelector
         : [ Expr ]
         | ArraySelector [ Expr ]
         ;

Type
         : int
         | float
         | string
         | datetime
         ;

Cont
         :                                /* nothing */
         | tNEWLINE
         ;

Term
         : tNEWLINE
         | ;
         ;

#tokens:
tAND_ASSIGN: &=                           (reserved)
tCASE: case                               (reserved)
tDEC_OP: --                               (reserved)
tDEFAULT: default                         (reserved)
tEOF: end-of-file
tINC_OP: ++                               (reserved)
tNEWLINE: newline character
tOR_ASSIGN: |=                            (reserved)
tSWITCH: switch                           (reserved)
tUSED_OP: used | ?                        (reserved)
tXOR_ASSIGN: ^=                           (reserved)
```

Appendix F

EuroSim files and formats

In this appendix an overview is given of the various files which are used and created by EuroSim. Also, for a number of files, their format is given.

F.1 EuroSim project files

In this section, each of the files which can be part of a EuroSim project is described briefly. Files used in a project can be identified by their extension:

Extension(s)	**Short description**
`adb`	Ada body source file.
`ads`	Ada spec source file.
`c`	C source file.
`cat`	SMP2 catalog(ue) file.
`dict`	Data dictionary; this derived file contains all API information for the simulator. It is generated by the Model Editor.
`EsimJournal.txt`	Human readable journal file; this file contains the logging of a simulation run.
`EsimJournal.xml`	Machine readable journal file; this file contains the logging of a simulation run.
`exe`	Simulator executable; this derived file is generated by the Model Editor.
`exports`	Exports file; contains variable nodes exported to simulation clients.
`f`, `F`	Fortran source file.
`h`	C header file.
`init`	Initial condition; this file contains initial conditions for a simulator. It is generated by the initial condition editor, which is integrated in the Simulation Controller.
`make`	Model makefile; this derived file controls the model building and is generated by the Model Editor.
`md`	Model Description file; Describes which variables of a model should be copied to the datapool. To edit a model description file, start the Model Description Editor (from the Model Editor).
`mdl`	Scenario file; this file contains an MDL scenario containing monitor (obsolescent), recording and stimuli definitions. It is generated by the Simulation Controller.

Extension(s)	Short description
`mmi`	Man-Machine Interface definition; this file describes the contents of an MMI tab page in the Simulation Controller. The contents consists of one or more monitors. This file replaces the use of monitors in the scenario (`mdl`) file.
`model`	Model file; contains all components for a simulator. To edit a model file, start the Model Editor.
`px`	Parameter Exchange file; Describes exchanges of data in the datapool. To edit a parameter exchange file, start the Parameter Exchange Editor.
`rec`	Recording file; this file contains data written by recording actions in the corresponding EuroSim scenario.
`sched`	Schedule file; contains all timing information for a simulator. The following files are referenced: the model and zero or more Parameter Exchange files. To edit a schedule, start the Schedule Editor.
`sim`	Simulation Definition file; contains references to all files needed to create and run a successful simulation. The following files are referenced: the model, the schedule, the optional exports, zero or more scenario files, initial condition files, MMI definitions or User Defined Program files. To edit Simulation Definition, start the Simulation Controller.
`snap`	Snap shot file; this file contains an full image of all API variables of an EuroSim simulator.
`timings`	Timings file; contains timings made during a simulation run. Can be imported by the Schedule Editor.
`tr`	Test result file; this file contains a list of all recordings performed by the corresponding EuroSim scenario.
`usr`	User Defined Program; contains all data necessary to launch a user defined program as client of the simulator.

The `tr`, `rec`, `timings`, `EsimJournal.txt` and `EsimJournal.xml` files are stored in directories representing the date and time of the simulation. The `exe` and `dict` files are created in a temporary directory that is made up of the basename of the model file and extension of the operating system (f.i. MyModel.Linux). The personal `.modelrc` file should be in the users home directory. All other files are in user-specified directories.

F.2 EuroSim Configuration file format

Most of the tunable settings of the EuroSim tools are controlled by settings in the system-wide configuration file which is stored in the file: `$EFOROOT/etc/esim_conf`.
If a user wants to have settings differently from these system-wide settings, he can copy the file `$EFOROOT/etc/esim_conf` to his home directory. At startup, the system-wide configuration file is read first, followed by the user's configuration file (if available).
Please note that a personal configuration file overrides any system-wide settings, so it is best only to include those settings that are actually changed.
The EuroSim configuration file is divided into two sections, the first section contains key-value pairs, the second section contains file type settings. Comment-lines are started with the # character.

F.2.1 Keys

Keys are defined with the format:

```
<key> = <value string>
```

The following keys are currently used by EuroSim:

UndoHistory
(the number of commands to remember for undo)

MakeCommand
(the command used to call GNU-make)

EuroSimOnlineHelp
(location of the help index)

ProjectManagerOnlineHelp
(location of Project Manager help)

ModelEditorOnlineHelp
(location of Model Editor help)

ModelDescriptionEditorOnlineHelp
(location of Model Description Editor help)

ParameterExchangeEditorOnlineHelp
(location of Parameter Exchange Editor help)

ScheduleEditorOnlineHelp
(location of Schedule Editor help)

SimulationCtrlOnlineHelp
(location of Simulation Controller help)

TestAnalyzerOnlineHelp
(location of Test Analyzer help)

F.2.2 File types

The definition of file types starts after the keyword "`FileTypes:`".
The format for file type entries:

```
<ID-string> : <description> : <extensions> : <editor cmd> : \
<viewer cmd> : <icon>
```

ID-string
uniquely identifying string

description
short description of the file type

extensions
the file extensions for the file type (comma separated)

editor cmd
the command used to edit the file

viewer cmd
the command used for read-only access to the file

icon the icon for the file type

The `<ID-string>` is mandatory, the other settings are optional. As an example follows an entry for the Simulation Defintion file type:

```
SIM_FILE : Simulation Definition : sim : SimulationCtrl : :
```

(Note: On the Windows NT platform, the EuroSim utility "`open.exe`" can be specified as an editor/viewer command to call the default editor defined under Windows NT.)

F.3 Recorder file format

The files written by the MDL `record` command and the files read by the MDL `stimulate` command both have the same file format.
Each file can contain input/output data for a number of variables. The number of variables in a particular file is stated at the beginning of the file. Following the line denoting the number of variables, is a set of lines, one for each variable, stating the variable name, variable type and variable dimension. The `<type>` field in the header is a basic type as defined in the C language, FORTRAN or Ada.

```
[Mission: <missionname.mdl>]
[Record size: <number of bytes>]
[Dict: <dictname.dict>]
[SimTime: <simtime_varname>]
[TimeFormat: relative/UTC]
Number of variables: <number>
{<variable_path> <type> {<variable_path_dimension>}}
```

Figure F.1: Syntax of EuroSim recording files.

Following these definitions is a set of lines, one for each input timepoint, stating the stimuli data to be inserted, or register data generated-, for each of the variables. The order of the values of the variables is the same as the definitions given for the variables.
The files all contain binary data for the `<variable_value>` records of the variable values. The headers of the files are in ASCII. In Figure F.1 (part of) the syntax definition is shown. When the file is generated by the `record` command, the first variable/column in the file will always be the simulation time variable[1]. Each invocation of the `record` command results in one record of variable values (see example in Figure F.2).

```
Mission: Demo48hr.mdl
Record size: 20
Dict: Demo48hr.dict
SimTime: /simulation_time
TimeFormat: relative
Number of variables: 3
/simulation_time: double
/BouncingBall/ballF77.f/balf77/ballvar$height: float
/BouncingBall/ballC.c/ballC/Velocity: double
```

Figure F.2: An example of a EuroSim recorder file.

The naming conventions for EuroSim recorder files are the following:

- for the files read and processed by the stimulation process any file name can be specified with the MDL `stimulate` command.
- for the files generated by the recording process a filename can be specified in the MDL `record` command[2], *or*
- for the files generated by the `record` command, when no file name is specified in the MDL `record` command, a file name is generated[3] with the name rec`-X-1.rec`.

[1]The variable for the simulation time can be specified by an environment variable.
[2]This way registration to a named file, and subsequent stimulation from a named file is possible within the same simulation run. For named registration the user should use `record "filename"` in MDL, for ”blind“ unnamed registration `record` suffices.
[3]Note that when the user changes an action containing registration commands the original registration file produced may be overwritten.

F.4 The test results file

The data recording process produces an `index` file in which all recorded Application model variables names are logged, including the name of the file where their values can be found. In Figure F.2 an example of an `index` file is shown. This file can be used to get a quick overview/index of the various variables recorded to disk during real-time simulation. It is meant to be used during off-line analysis of the recorded data.
The name of the index file is derived from the name of the ready-to-run simulator executable filename. If that is `SUM.exe` then the index file will get the filename `SUM.exe.tr`.

```
Filename                Variable
SateliteDecayTest.rec /simulation_time
SateliteDecayTest.rec /Altitude/altitude
SateliteDecayTest.rec /Thruster/thrusterOnOff
SateliteDecayTest.rec /Altitude/decaySpeed
```

Figure F.3: An example of a test results file.

F.5 Exports file format

The exports file (which has as name *modelname*`.exports`) describes which part of the EuroSim data dictionary may be accessed by external (non-EuroSim) simulators. For each part that should be accessible for external simulators, one can indicate how it can be accessed (read, write, or both) and by whom.
The exports file consists of a number of lines, each line describing one part of the data dictionary that may be exported. Empty lines and lines beginning with `#` are ignored. Data following a `#` is considered to be a comment. Each non-empty line has the following layout:

```
{path id mode users}
```

Where `path` is the path to the data dictionary which should be exported, `id` is the name under which this path should be exported, `mode` is the operation that can be performed (`R`, `W` or `RW`) and `users` is a list of clients that may request access.
The given path that is exported means that every subtree or variable that is located underneath that path may be requested in a view. A simple way therefore to export every variable is to export the `/`. The `id` under which the path is exported is the name which the external simulator must use in his access request.
The access mode `RW` is not yet implemented. However, it is possible to add separate read and write export lines.
When no users are specified the export operation is valid for all users. For more information, see also the `exports(4)` man page of EuroSim, and chapter Chapter 18.
Example exports file:

```
#
# Example file
#
/space/station/era    era       R
/space/stars          stars     RW
/space/rockets/ariane esarocket W
```

F.6 Initial Condition file format

The Initial Condition file format is either ASCII or binary. The extensions of these files are `.snap` or `.init`.
The file consists of a header section and a data section. Empty lines and lines starting with `#` in the header section are ignored as comment lines. However, when the rest of the line following a `#` character contains valid keyword/value pairs, it is interpreted. Keyword/value line have the form:

```
#keyword = value
```

Valid keywords are:

comment
: the comment that was passed when the file was written. May be omitted.

format
: either ASCII or binary. When omitted the file is interpreted as being ASCII.

simtime
: the simulation time at the time the snapshot was taken.

dict
: the EuroSim data dictionary file from which this file was written. The path to the data dictionary is relative to the place where it can be found. May be omitted.

reference
: an optional version control reference for the state of the model this file's data dictionary was generated from.

Any other keywords can be generated by `dictdump`, or by the user, but they are not interpreted. Every initial condition or snapshot file written by EuroSim also contains a comment line indicating the type of snapshot or initial condition file written. It is either:

```
# contains only differences wrt dict default values
```

or

```
# contains all current dict values at <date>
```

which indicates whether it is a partial snapshot (or initial condition) file or a complete snapshot containing all the variables in the data dictionary.
When the format of the file is binary there is at least one mandatory empty line following the header.
The data section of a binary file contains records for each data dictionary symbol as follows:

```
{ symbol_length+1, symbol, value_length, value }
```

where the symbols are fully qualified data dictionary paths and the values for the symbols are of course in 'binary' form (no formatting).
When the format of the file is ASCII the records of the data section look like:

```
{ "InitialCondition: ", symbol, "=", value }
```

Again the symbols are fully qualified data dictionary paths, the values for the symbols are formatted. The records may extend several lines but the carriage return '`\n`' is then escaped with a \ backslash, so in there is in principle one record per line.
The following example shows a typical layout of a full (ASCII) initial condition file:

```
# EuroSim initial condition file
# version = @(#)Header: dumpfile
# dict    = thermo.dict
# comment = complete ascii dump
# format  = ascii
# contains all current dict values at Mon Jan 27 14:15:24 1997
#
InitialCondition: /thermo.f/thermo$celltemp = "{ { 0, 0, 0},\
{ 0, 0, 0}, { 0, 0, 0}, { 0, 0, 0}}"
InitialCondition: /thermo.f/initthermo/thermo$capa = "{ { 0,\
0, 0}, { 0, 0, 0}, { 0, 0, 0}, { 0, 0, 0}}"
InitialCondition: /thermo.f/initthermo/thermo$condfac = "0"
InitialCondition: /thermo.f/initthermo/thermo$emisfac = "0"
```

F.7 Simulation Definition file format

The format of the `.sim` file (and also of the `.mmi` and `.usr` files) is a simple keyword-value format:

```
keyword value;
```

where `value` is either a number or a text between double quotes. To embed a double quote in the text you have to prefix it with a backslash. To embed a backslash in the text you also have to prefix it with a backslash. Examples:

```
foo 1;
bar "text example";
escape "quote \" backslash \\";
```

A keyword can also start a nested set of keyword-value pairs. Example:

```
nested_keyword {
 key1 value1;
 key2 value2;
}
```

The simulation definition file supports the following keywords:

version the version number of the file format

server the server to use for the simulator

resultsPath
the directory where the result files are stored.

createSubDir
if `1`, then create <date>/<time>subdirectories in the `resultsPath` directory and store the result files there. If `0`, then do not create these subdirectories.

model start a nested section for a model file. See below for valid keywords.

schedule
start a nested section for a schedule file. See below for valid keywords.

export start a nested section for an exports file. See below for valid keywords.

mdl start a nested section for a scenario file. See below for valid keywords. This keyword can be used more than once.

mmi start a nested section for an mmi file. See below for valid keywords. This keyword can be used more than once.

usr start a nested section for an usr file. See below for valid keywords. This keyword can be used more than once.

ic start a nested section for an initial condition file. See below for valid keywords. This keyword can be used more than once.

Valid keywords for the `model`, `schedule`, `export` and `usr` nested sections:

path the path of the file

required
the required version of the file

Valid keywords for the `mdl` nested section:

path the path of the scenario file

required
: the required version of the file

caption
: the caption of the corresponding tab page

active
: if `1`, then the scenario is active, otherwise it is inactive

iconView
: if `1`, then represent the scenario using an iconview, if `0`, then the scenario is represented as a treeview.

Valid keywords for the `mmi` nested section:

path
: the path of the `mmi` file

required
: the required version of the file

caption
: the caption of the corresponding tab page Valid keywords for the `ic` nested section:

path
: the path of the initial condition file

required
: the required version of the initial condition file

active
: if `1`, then the initial condition is active, otherwise it is inactive

Syntax

```
SIM
 /* Simulation Definition file */
 : keyvals
 | tEOF
 ;
keyvals
 : keyval
 | keyvals keyval
 ;
keyval
 : server string ;
 | version numeric ;
 | model { file_keyvals }
 | schedule { file_keyvals }
 | export { file_keyvals }
 | usr { file_keyvals }
 | ic { ic_keyvals }
 | mdl { mdl_keyvals }
 | mmi { mmi_keyvals }
 ;
file_keyvals
 : file_keyval
 | file_keyvals file_keyval
 ;
file_keyval
 : path string ;
 | required string ;
 ;
ic_keyvals
```

```
 : ic_keyval
 | ic_keyvals ic_keyval
 ;
ic_keyval
 : path string ;
 | required string ;
 | active numeric ;
 ;
mdl_keyvals
 : mdl_keyval
 | mdl_keyvals mdl_keyval
 ;
mdl_keyval
 : path string ;
 | caption string ;
 | required string ;
 | active numeric ;
 | iconView numeric ;
 ;
mmi_keyvals
 : mmi_keyval
 | mmi_keyvals mmi_keyval
 ;
mmi_keyval
 : path string ;
 | required string ;
 | caption string ;
 ;
```

F.8 MMI file format

The format is identical to the simulation definition. The purpose of an MMI file is to define monitors and action buttons on a tab page. The following keywords are valid for the MMI format:

version the version number of the file format (should be `2`)

monitor start a nested section for a monitor definition. See below for valid keywords. This keyword can be used more than once.

Valid keywords for the `monitor` nested section:

name the caption of the monitor or action button

mdl the scenario file containing the action used by the action button. Use an empty string if not relevant.

action the action executed or disabled/enabled by the action button. Use an empty string if not relevant.

monitorType
the type of the monitor:

Type	Description
0	Alpha numerical monitor
1	Plot against the simulation time

Table F.2: Monitor Types

Type	**Description**
2	Plot against the wall clock time
3	Plot against another variable
4	Action button

Table F.2: Monitor Types

history the maximum number of data points that are used for the plot.

left the position of the left edge of the monitor in pixels

top the position of the top edge of the monitor in pixels

width the width of the monitor in pixels

height the height of the monitor in pixels

manualScalingX
if `1`, then the X-axis has a fixed range, otherwise the X-axis scales automatically.

xMin the minimum value of the X-axis

xMax the maximum value of the X-axis

manualScalingY
if `1`, then the Y-axis has a fixed range, otherwise the Y-axis scales automatically.

yMin the minimum value of the Y-axis

yMax the maximum value of the Y-axis

var start a nested section for a variable definition. See below for valid keywords. This keyword can be used more than once.

Valid keywords for the `var` nested section:

name the variable to monitor

showLine
if `1`, then draw the line connecting two data points.

lineColor
the color of the line. It is the decimal representation of the hexadecimal RGB value 0xR-RGGBB.

symbol the symbol to use for a datapoint.

Value	**Description**
0	No symbol
1	Ellipse
2	Rectangle
3	Diamond
5	Down triangle
6	Up triangle
7	Left triangle

Table F.3: Available Symbols

Value	Description
8	Right triangle
9	Cross
10	X-Cross

Table F.3: Available Symbols

Note that value 4 is not used.

symbolColor
: the color of the symbol. It is the decimal representation of the hexadecimal RGB value 0xRRGGBB.

readOnly
: if `1`, then this variable is read only.

Syntax

```
MMI
 /* Man-Machine Interface file */
 : keyvals
 | tEOF
 ;
keyvals
 : keyval
 | keyvals keyval
 ;
keyval
 : monitor { monitor_keyvals }
 | version numeric ;
 ;
monitor_keyvals
 : monitor_keyval
 | monitor_keyvals monitor_keyval
 ;
monitor_keyval
 : var { var_keyvals }
| name string ;
| mdl string ;
| action string ;
| monitorType numeric ;
| history numeric ;
| left numeric ;
| top numeric ;
| width numeric ;
| height numeric ;
| manualScalingX numeric ;
| xMin numeric ;
| xMax numeric ;
| manualScalingY numeric ;
| yMin numeric ;
| yMax numeric ;
;
var_keyvals
```

```
  : var_keyval
  | var_keyvals var_keyval
  ;
var_keyval
  : name string ;
| showLine numeric ;
| lineColor numeric ;
| symbol numeric ;
| symbolColor numeric ;
| readOnly numeric ;
;
```

F.9 User Program Definition file format

The format is identical to the simulation definition. The purpose of a `.usr` file is to specify a program that can be used to connect to a running simulator. The following keyword is valid for the `.usr` format:

def the specification of the program and its arguments. Note that the sequence `%h` is replaced with the hostname of the running simulator and the sequence `%c` is replaced with the preferred connection number.

Syntax

```
USR
  /* User Program Definition file */
  : keyvals
  | tEOF
  ;
keyvals
  : keyval
  | keyvals keyval
  ;
keyval
  : def string ;
  ;
```

Appendix G

API header layout

This appendix contains the lay-out of the API headers, as they are generated by EuroSim for C and Fortran model code. As EuroSim does not generate API headers for Ada-95 model code, the information in this appendix can be used to create API headers for Ada-95 model code by hand.
The API header is contained in a comment block at the top of the source code
(i.e. between `/* */` in C, on lines starting with `C` in Fortran and on lines starting with `--` in Ada-95). In Ada-95 and Fortran, make sure that if the original source code started with a comment block, that there is an empty line between the API header and the source code comments.
Each API header consists of the following four keywords (see Section 2.5 for more information):

- `'Global_State_Variables`
- `'Global_Input_Variables`
- `'Global_Output_Variables`
- `'Entry_Point`

The first three keywords are used to describe the variables in the source code, and the last keyword is used to describe the entrypoints. The first keyword is used once per source file, the last three once per entrypoint.
Each keyword is preceded by a straight quote.

G.1 `'Global_State_Variables`

Global state variables are the variables which are used in the current source file only, and should not be seen by other source files.
The syntax of the keyword is:

`'Global_State_Variables` *VariableType VariableName* : *Attributes*

The *VariableType* and *VariableName* are as they are defined in the source file. The *Attributes* can be zero or more of the attributes described below. If more than one attribute is used, they should be separated by spaces or newlines. If more than one variable is defined with the keyword, each *VariableType VariableName* : *Attributes* set should be separated by commas.

- `UNIT="`*text*`"`

This defines *text* as the unit of the variable. The string *text* can be any string.

- `DESCRIPTION="`*text*`"`

This defines a string *text* which is used as description of the variable.

- `PARAMETER` or `RO`

No additional information. It defines a variable as ‘parameter’, meaning that EuroSim should not allow the value of the variable to be changed during a simulation (only during initialization).

- `INIT="`*value*`"`

This defines *value* as the initial value for the variable. *value* should be in the correct syntax for the associated variable.

- `MIN="`*value*`"`
- `MAX="`*value*`"`

These two define the minimum and maximum values of the variable. *value* should be in the correct syntax for the associated variable.

G.2 `'Global_Input_Variables`

This keyword is used to define the variables that are used by the current source file, and which are set to a value by another source file. The syntax of the keyword is the same as for global state variables.

G.3 `'Global_Output_Variables`

This keyword is used to define the variables that are used by other source files, and which are set to a value by the current source file. The syntax of the keyword is the same as for global state variables.

G.4 `'Entry_Point`

This keyword is used once per function/procedure that has to be available for the scheduler. See Appendix H for more information on restrictions on functions/procedures to be used as entrypoints.
The syntax of the keyword is:

`'Entry_Point` *FunctionName* `: DESCRIPTION="`*Description*`"`

G.5 Publishing of variables

It is also possible to ‘publish’ variables from the data dictionary. There are several functions that set the address where a variable or entrypoint in a certain data dictionary is stored, thus making it accessible from the outside. This is useful for people who want to make their own model interfaces.
The publish functions are divided in two categories, a function to get the runtime data dictionary and functions to publish data variables and entrypoints in a data dictionary.

G.5.1 Function to get the runtime data dictionary

When a EuroSim simulation application program needs access to the runtime data dictionary it must call `esimDict(void)`. This function returns a pointer to the runtime data dictionary (`DICT*`) and is defined in the header file `esimDict.h`.

G.5.2 Functions to publish data variables and entrypoints in a data dictionary

`dictPublish(DICT *dict, const char *name, const void *address)` sets the address of the variable specified by *name* in the data dictionary specified by *dict* to *address*. This function can be called from C or Ada.

`dictpublish_(DICT *dict, const char *name, const void *address, int namelen)` is the Fortran wrapper for `dictPublish`. It has an extra parameter with the length of the *name* parameter. This is required by the calling convention of Fortran functions.

`dictPubEntry(DICT *dict, const char *name, EntryPtr address)` sets the function address of the entrypoint specified by *name* in the runtime data dictionary to *address*. This function can be called from C or Ada.

`dictpubentry_(DICT *dict, const char *name, EntryPtr address, int namelen)` is the Fortran wrapper for `dictPubEntry`. It has an extra parameter with the length of the *name* parameter. This is required by the calling convention of Fortran. functions.

The prototypes for these functions can be found in `DictPublish.h`.

G.6 Example API header

G.6.1 Example in C

As an example, the API header from the Thruster.c file used in the case study is shown below (see Section 4.5 for the source code and the API information).

```
/*
'Entry_Point Thruster:
   DESCRIPTION="The thruster brings the satellite to"
   " the correct altitude."
   'Global_Input_Variables
      int lowerAltitudeLimit:
         UNIT="km"
         DESCRIPTION="Below this limit, the thruster must"
         " be turned on."
         INIT="210"
         MIN="0"
         MAX="1000",
      int sateliteAscentSpeed:
         UNIT="km/h"
         DESCRIPTION="The ascent speed of the satellite."
         INIT="10"
         MIN="1"
         MAX="200",
      int thrusterOnOff:
         UNIT="On/Off"
         DESCRIPTION="Indicates whether the thruster is"
         " on or off."
         INIT="1"
         MIN="0"
         MAX="1",
      int upperAltitudeLimit:
         UNIT="km"
         DESCRIPTION="The upper limit at which the thrust"
         "er is to be switched of."
         INIT="280"
         MIN="0"
         MAX="1000"
   'Global_Output_Variables
```

```
      int thrusterOnOff:
         UNIT="On/Off"
         DESCRIPTION="Indicates whether the thruster is"
         " on or off."
         INIT="1"
         MIN="0"
         MAX="1"
*/
```

Note that there is no restriction on line length for the API headers, but that the API Editor generates no lines longer than 80 characters. This is done to ensure good readability on most terminals.
Also note that variables which act both as input as well as output variables are defined twice in the API header.

G.6.2 Example in Ada-95

```
---------------------------------------------------------
--
-- Name:    ball.adb
-- Type:    Ada-95 implementation.
--
-- Author: John Graat (NLR).
-- Date:    19961125
-- Changes: none
--
--
-- Purpose: Model for the Simulation of a Bouncing Ball.
--
--          The Bouncing Ball describes a ball that is thrown
--          straight-up from the ground with an initial velocity
--          or dropped from an initial height.
--          In the absence of friction, the ball should reach
--          exactly the same maximum height time and time again.
--          The ball is described as a mass point.
--
-- Parameters: GRAVITY Gravitation constant [m/s2]
--
-- State:   Height Height of the ball above the ground [m].
--          Velocity Velocity of the ball [m/s].
--
-- Additional: DeltaT Time Step for the Model.
--          LoadLoop Loop counter to increase computation time.
--          Duration Duration of the Ball Model.
--
-- Remark: The mass of the ball has mplicitly been set to 1 [kg].
--
-- API Header required for the correct Data Dictionary:
--
--     'Entry_Point ball.Ball:
--       DESCRIPTION="Computation of one time step of the ball"
--                "."
--        'Global_Input_Variables
--          Long_Float ball.deltat:
--                UNIT="s"
--                DESCRIPTION="Time step for the Ball Sub-Model."
--                MIN="0"
--                MAX="1",
--          Long_Float ball.height:
--                UNIT="m"
--                 DESCRIPTION="Height of the ball."
--                 MIN="0"
--                 MAX="100",
--           Integer ball.loadloop:
--                 UNIT="-"
--                 DESCRIPTION="Loop counter to increase load."
--                 MIN="0",
--           Long_Float ball.velocity:
--                 UNIT="m/s"
--                 DESCRIPTION="Velocity of the Ball."
--         'Global_Output_Variables
--           Long_Float ball.deltat,
--           Long_Float ball.height,
--           Long_Float ball.velocity,
--           Long_Float ball.duration:
--                 DESCRIPTION="Duration of the Ball Model."
--
-----------------------------------------------------------------

with integr;
with esim;
use esim;

package body Ball is

  GRAVITY : constant Long_Float := 9.80664999;

  -- Global variables of the Bouncing Ball
  -- Actual declaration of these variables can be found in ball.ads
  -- Height, Velocity, DeltaT : Long_Float;
  -- Duration           : Long_Float;
  -- LoadLoop           : Integer;

  procedure Ball is
    -- Local Variables of the Bouncing Ball
    State, Dot    : Integr.Vector;
    Rate, Fine    : Long_Float;
    Loopcnt       : Integer;
    Start, Stop   : Long_Float;

    begin
      -- Get the Start time from the Wall Clock.
      Start := esimGetWallclocktime;

      -- Get DeltaT Time from the EuroSim Tool.
      Rate := EsimGetTaskrate;
      DeltaT := 1.000/Rate;
      Fine := DeltaT/Long_Float(100);

      for Counter in 1 .. 100 loop
        State(1) := Height;
        State(2) := Velocity;
        Dot(1) := Velocity;
        Dot(2) := -GRAVITY;

        -- Forward Euler Integration.
        Integr.intEulerADA( State, Dot, 2, Fine );

        -- Check on events, e.g. Ball touches the ground.
        if State(1) < 0.0 then
```

```
          State(2) := -State(2);
        end if;

        Height := State(1);
        Velocity := State(2);
      end loop;

      Loopcnt := 0;

      -- Loop to increase the computation time of the model.
      for Counter in 1..LoadLoop loop
        Loopcnt := Loopcnt + 1;
      end loop;

      -- Get Stop time from the Wall Clock and calculate Duration.
      Stop   := esimGetWallclocktime;
      Duration := Stop - Start;
  end Ball;
end Ball;
```

Appendix H

Programming language limitations

H.1 Generic limitations

Model code should follow a set of rules when it is to be used in EuroSim. The rules are:

- Entrypoints should have no return value.
- Entrypoints should have no calling arguments/parameters (functions not used as entrypoints do not have this restriction). When calling arguments or parameters are needed they should be defined through one of two methods (of which the first one is recommended):
  1. Define global variables through an API as 'virtual' arguments/parameters.
  2. Encapsulate a function with arguments in a function which complies to the guidelines; this function can then call the function with arguments.
- If the entrypoint is used in the real-time domain it is not allowed to use any operating system call (`open`, `printf`, etc...). This is because operating system calls do not have deterministic execution times. Calls which are allowed are the services provided by EuroSim. See Appendix D for details on the EuroSim services.
- The entrypoint must not create a deadlock (i.e. waiting on a resource not available for some (undefined) time).
- No names should be used which conflict with one of the internal EuroSim functions. Refer to the file `$EFOROOT/etc/reserved-words.txt` for the complete list of reserved words.
- Only variables with a memory address that is fixed at load time can be used as API variables[1].

The operation must not make use of a locking mechanism (semaphores) to establish mutual exclusion of a common defined variable. This should be done using an asynchronous store (see Section 11.3).
During real-time simulation, the size of the system stack cannot change. Therefore, care should be taken with model code which allocates large data structures on the stack.
When combining programming languages in one model (e.g. C and Fortran), there are a number of rules to keep in consideration with respect to variable and function naming. Refer to the programming language documentation for more information. For an example, see Section 4.6.

H.2 C limitations

Unnamed structures, unions and bitfields cannot be used as API variables.

[1]There is one exception: static variables declared within a C function have a load time fixed address but are not accessible by EuroSim. No implementation of such access is possible without violating the rule that EuroSim should not modify source code files.

H.3 Fortran limitations

Because Fortran lacks the `extern` keyword as available in C, the ‘owner’ of a variable is not known to the Fortran compiler. Therefore, variables are declared in more than one Fortran source file. However, for EuroSim purposes, the API information for a variable should only be in *one* API header. The user should therefore make sure that a variable which is declared in more than one source file, should only be added to the API header of one of those files.

H.4 Ada-95 limitations

Although EuroSim does support the use of Ada-95 (except on the Windows NT platform) for the development of model code, the support is not at the same level as for C and Fortran. This is mostly due to the complexity of the Ada-95 language. The main difference with the use of C and Fortran code is that the API Editor does currently not support parsing of Ada-95 code. This means that any API headers have to be entered by hand to the source code. See Appendix G for details on the layout of the API headers, as well as an example Ada-95 header. Also, EuroSim currently only supports the use of the “GNAT” Ada-95 compiler. In this section, the limitations of the use of Ada-95 are described.

H.4.1 Compilation

The GNAT compiler allows only one compilation unit per file. The `gnatchop` utility can be used to split the files. A body should be contained in a `.adb` file, and specifications should be in `.ads` files. If the package name `example` is given in a `with` clause, the compiler will look for `example.ads`. Filenames are mapped to lowercase, so the file `Example.ads` will not be found.

H.4.2 Variables

Only variables which have a fixed address (as specified by the Ada-95 ‘Address’ attribute) can be used as global variables within EuroSim. Variables that are to be used as globals must be made visible to the generated publish procedure. Therefore they must be put in a subprogram or package specification, so that they can be accessed by means of the `with` clause.
When two packages define a variable with the same name, the names should be fully qualified in the data dictionary (i.e. with the package name), otherwise the connection between variables and their compilation subunits would be lost.
If Ada-95 code is mixed with C and/or Fortran code, the model developer has to get the bindings of variable and entry names correct themselves. An entity `name` that appears in a library package is accessible from C as `package__name` (two underscores). If the entity appears outside a package, its name will be prefixed with `_ada_`.

H.4.3 Entrypoints

Ada-95 procedures without arguments can be used as entrypoints. In contrast with the global variables, they will not be referenced from generated Ada-95 publish code. However, they will be called from C code that is generated using information in the data dictionary, so the name in the data dictionary should correspond to the generated name in the object file.
Since entrypoints cannot have arguments, they cannot be overloaded.

H.4.4 Types

Generic packages cannot have API headers, because each instantiation would also have to instantiate a new API header. The API header has no support for generic types. If an instantiation of a generic package is made, the user has to perform the necessary parameter substitution himself.
User defined types are not supported by EuroSim.

H.4.5 Tasks

Since the EuroSim environment supplies its own task mechanism, the Ada-95 task and exception mechanism and associated commands (e.g. `select`, `delay`) should not be used.

H.4.6 Debugging support

As the `dbx` debugger on IRIX does not support Ada-95, if Ada-95 debugging support is needed, an Ada-95 debugger with support for mixed language environments (C and Fortran) should be used (e.g. gdb).

H.4.7 Real time aspects

The timing of Ada-95 routines may be less predictable than the timing for C and Fortran, due to the dynamic allocation of variables.

Appendix I

HLA extension: EsimRTI[1]

I.1 Introduction

The EsimRTI is a HLA interface for EuroSim. The EsimRTI is a real-time layer on top of the RTI that provides the RTI-services to an application in a real-time manner with a 'C' application programmer's interface.
EuroSim models that can use the EsimRTI must be composed from C source files. It is possible to include Fortran 77 source files in the EuroSim model but the EsimRTI API must be called from the C source files.

I.1.1 EsimRTI usage with EuroSim

The list of EuroSim capabilities has been extended with an entry for EsimRTI and RTI (this allows the EuroSim user to select that capability in the Model Editor: *Tools:Set Build Options*.

I.1.2 EsimRTI usage without EuroSim

The functionality provided in the EsimRTI can be used without EuroSim as well. This stand-alone use of the EsimRTI library is available in two versions:

- A single-threaded library (libEsimRTIsts.a).
- A multi-threaded library (libEsimRTImts.a).

To use the stand-alone versions of the EsimRTI libraries compile your source code with the preferred preprocessor defines and link with the appropriate EsimRTI library. The relevant preprocessor defines are `ESIMRTI_STANDALONE` (mandatory) and `ESIMRTI_MULTI_THREADED` (optional).
The relevant include files are located in `$EFOROOT/include/RTI`.
The libraries are located in `$EFOROOT/lib32/esim`.
The functions esimMalloc and esimFree are replaced with malloc and free for the EsimRTI usage without EuroSim. The functions esimMessage, esimWarning, esimError, esimFatal, esimReport and esimSetState are replaced with appropriate printf statements EsimRTI usage without EuroSim.

I.1.3 Running

To execute a EuroSim model or to run a program that uses the EsimRTI, the rtiexec (which can be found in `$RTI_HOME/bin/IRIX-6.5-n32/`) should be started somewhere on the network. IP-multicasts should be able to reach this rtiexec-host. The appropriate fed-file should be present in the `$RTI_CONFIG` directory.

[1]Not supported in the Linux and Windows NT version.

I.1.4 Memory use of EuroSim models with EsimRTI extension

I.1.4.1 Introduction

To ensure hard real time execution of EuroSim models EuroSim loads the complete executable and libraries (static and shared) into memory. The user limit `maxlkmem` controls the amount of memory that can be locked to load the executable and libraries into.
With the use of the EsimRTI and the RTI the amount of memory required for text and data to run a EuroSim model increases significantly. The size of the static EsimRTI library is 3.1 MB (libEsimRTI.a). The size of the shared library is 2.6 MB (libesRTI.so). The size of the RTI (1.3v6) libraries is 16.6 MB (libRTI.so) and 0.1 MB (libfedtime.so). The increased total size of the executable and libraries might exceed the user limit `maxlkmem` more easily.
Specifically the use of exceptions by the RTI can cause the user limit maxlkmem to be exceeded. In the EuroSim log file (`/var/adm/esimd.<hostname>.log`) this can be detected by occurrences of lines similar to: `Assertion failed in file '../../libC/lang_support/throw.cxx', line 1614`
The user limit `maxlkmem` can be changed according to the size of the executables (including the libraries) and the pagesize of the machine. The limit must be set to a higher value than the executable size divided by the pagesize:

- The size of the executable (including the libraries) can be determined with the following command: `size -4` *executable*
- The pagesize of the machine can be determined with the following command:

  ```
  sysconf PAGESIZE
  ```
- Calculate the minimum value for the `maxlkmem` limit executable size (in bytes) / page size. Note that the `maxlkmem` should not exceed half the amount of physical memory available. The amount of physical memory available can be determined with the `hinv` command.
- The user root can change the `maxlkmem` user setting with the following command sequence:

  ```
  systune -i

  Updates will be made to running system and /unix.install
  systune-> maxlkmem <sufficient # pages>
          maxlkmem = 2000 (0x7d0)
  Do you really want to change maxlkmem to <sufficient # pages> (0x<...>)? (y/n) y
  systune-> quit
  ```

I.1.4.2 Example

The steps to determine the amount of memory used by a EuroSim model described in the previous section are applied to the executables of the flywheel example (introduced elsewhere in this appendix).

- `size -4 <executable>` reports:

  ```
  .../esim-projects/flywheelengine$ size -4 *.exe
  engine.exe: 45056 + 12288 + 0 = 57344
  flywheel.exe: 45056 + 12288 + 0 = 57344
  flywheelengine.exe: 20480 + 8192 + 0 = 28672
  ```
- On the O200 that was used to developed the EsimRTI `sysconf PAGESIZE` reports:

  ```
  .../esim-projects/flywheelengine$ sysconf PAGESIZE
  16384
  ```
- The minimum value for `maxlkmem` is calculated:

```
engine.exe 57344000 / 16384 = 3500
flywheel.exe: 57344000 / 16384 = 3500
flywheelengine.exe: 28672000 / 16384 = 1750
```

- On the O200 that was used to develope the EsimRTI the `hinv` command reports the amount of physical memory available (bold in the following output):

```
.../esim-projects/flywheelengine$ hinv
4 225 MHZ IP27 Processors
CPU: MIPS R10000 Processor Chip Revision:  3.4
FPU: MIPS R10010 Floating Point Chip Revision:  0.0
Main memory size:  512 Mbytes
Instruction cache size:  32 Kbytes
Data cache size:  32 Kbytes
Secondary unified instruction/data cache size:  2 Mbytes
Integral SCSI controller 3:  Version QL1040B (rev.  2), single ended
Disk drive:  unit 1 on SCSI controller 3
Integral SCSI controller 2:  Version QL1040B (rev.  2), single ended
WORM: unit 5 on SCSI controller 2
Integral SCSI controller 1:  Version QL1040B (rev.  2), single ended
CDROM: unit 6 on SCSI controller 1
Integral SCSI controller 0:  Version QL1040B (rev.  2), single ended
Disk drive:  unit 1 on SCSI controller 0
Integral SCSI controller 4:  Version QL1040B (rev.  2), single ended
IOC3 serial port:  tty1
IOC3 serial port:  tty2
IOC3 serial port:  tty3
IOC3 serial port:  tty4
IOC3 parallel port:  plp1
IOC3 parallel port:  plp2
Integral Fast Ethernet:  ef0, version 1, module 1, slot io1, pci 2
Fast Ethernet:  ef1, version 1, module 2, slot MotherBoard, pci 2
Origin 200 base I/O, module 2 slot 2
Origin PCI XIO board, module 1 slot 7:  Revision 4
Origin 200 base I/O, module 1 slot 1
IOC3 external interrupts:  1
IOC3 external interrupts:  2
PCI card, bus 0, slot 5, Vendor 0x12e2, Device 0x4013
```

- A sufficient value for the `maxlkmem` setting would be 4000 for the above mentioned executables. This value would represent lock 65.5 MB of memory (4000 * 16384). On the O200 this represents 13% of the available physical memory.

- To change the setting for `maxlkmem` use `systune -i` (requires root privileges):

```
# systune -i
Updates will be made to running system and /unix.install

systune-> maxlkmem 4000
maxlkmem = 8192 (0x2000)
Do you really want to change maxlkmem to 4000 (0xfa0)? (y/n) y

systune-> quit
```

I.2 Implementation and usage notes

The following sections describe details on how the EsimRTI library can be used and a number of implementation aspects.

I.2.1 Direct and Buffered modes

I.2.1.1 Introduction

The EsimRTI can operate in two different modes: Direct and Buffered.

- In the Direct mode (EsimRTImodeDirect) any call into the RTI is directly processed by the RTI. Federate ambassador callbacks are executed when the EsimRTI is ticked.
- In the Buffered mode (EsimRTImodeBuffered) invocation of EsimRTIrtiAmbassador calls will only result in the queuing of the calls (in a call buffer). The calls will not be executed directly. Federate ambassador callbacks are queued as well and are executed when the EsimRTI is ticked.

The synchronous RTI calls are not available in buffered mode (an overview is presented elsewhere in this document).

I.2.1.2 Disadvantages of the use of the Direct mode

Using the direct processing of the RTI calls has the following disadvantages.

- The amount of time the RTI uses to process a call is not predictable.
- Although the RTI throws an exception when an application attempts to access the RTI concurrently (RTI calls can be made from multiple tasks executed in parallel) special precautions are required to prevent concurrent access to the RTI (and to ensure the call is processed).

I.2.1.3 Disadvantages of the use of the Buffered mode

The mechanism of buffered or deferred function calls (at least) has the following disadvantages.

- The perception the application has of the internal state of the RTI does not necessarily always match to the actual internal state of the RTI, for short periods the two may differ. This wrong perception can lead to inconsistencies in the delivery of messages to the application.
- The reason is of course that a call into the RTI is not processed directly by the RTI, and conversely, messages delivered by the RTI are not received directly by the application. Methods stored in the Federate callback buffer may conflict with the perception of the application of the internal state of the RTI, for instance: Suppose the application just divested the ownership of an object unconditionally (which means that from now on the federate no longer has any update responsibility), while the Federate callback buffer holds the request for the update of the attributes of the very same object. After ticking the EsimRTI, the application will receive the request, and, since it relies on the internal state of the EsimRTI, does no checking, and sends the updates of the attributes of the object. When the RTI processes this updateAttribute, it will generate the exception ObjectNotKnown.
- To ensure buffer consistency the operations on the buffer must be mutual exclusive. Mutex locks and release calls are used to implement this. Locking and releasing the mutex variable takes time of course.

I.2.1.4 Suggested usage

The easiest use of the EsimRTI is:

- To implement all RTI related tasks in *soft* real-time tasks.
- To use the direct mode during initialization and finalization.
- To use buffered mode during standby and execution.

I.2.2 Single and Multi threaded EsimRTI libraries

The EsimRTI library is available in three versions:

- EuroSim EsimRTI (single threaded).
- Stand-alone single-threaded.
- Stand-alone multi-threaded.

The differences between these versions is expressed in

- the presence or absence of a separate thread, and
- the functionality of the EsimRTIrtiAmbassadorTick function (the method that has replaced the RTI::RTIambassador.tick).

In the single-threaded versions of the EsimRTI library ticking the EsimRTI (using esimRTIrtiAmbassadorTick):

- delivers one (in Buffered mode) or more (in Direct mode) RTIambassador call(s) to the RTI;
- ticks the RTI; and
- delivers one (in Buffered mode) or more (in Direct mode) federateAmbassador callback(s) to the application;

In the multi-threaded version of the library a thread is created when the EsimRTIrtiAmbassador is created. The thread is suspended when created and when the EsimRTI mode is switched to Direct mode. The thread is resumed when the EsimRTI mode is switched to Buffered mode. The thread takes care of:

- flushing the queued RTIambassador calls; and
- ticking the RTI.

Ticking the RTI delivers federateAmbassador callbacks from the RTI to the EsimRTI library and queues them in a callback buffer. To deliver the queued callbacks the EsimRTI must be ticked using esimRTIrtiambassadorTick.
To use the multi-threaded version of the EsimRTI library (for stand-alone applications only) compile the source code of your application with `ESIMRTI_MULTI_THREADED` defined (using the `-D` compiler option) and link with the right library (libEsimRTImts.a). To compile the source code of your application outside the EuroSim environment `ESIMRTI_STANDALONE` must be defined (`-DESIMRTISTANDALONE`).

I.2.2.1 (Dis)advantages of the multi-threaded version of the EsimRTI library

Using the multi-threaded has disadvantages and advantages.

- An advantage is that the application developer does not have to worry about the delivery of the RTI calls to the RTI and the queuing of the federate ambassador callbacks from the RTI for the application (although the EsimRTI must be ticked to deliver the queued callbacks to the application).

- An advantage is that the queued RTI-calls are transferred to the RTI on a regular basis.
- A disadvantage is that with a high the number of RTI calls and/or federate ambassador callbacks the buffers might overflow because the application developer has little influence on the amount of processing the thread is allowed.

For these reasons the EuroSim version of the EsimRTI library is single-threaded. The esimRTIrtiAmbassadorTick method transfers the RTI calls (one or more depending on the EsimRTI mode) to the RTI, ticks the RTI and transfers the queued federate ambassador callbacks to the application (one or more depending on the EsimRTI mode).

I.2.2.2 Functionality of esimRTIrtiAmbassadorTick and esimRTIthread

All versions of the EsimRTI library include a function esimRTIrtiAmbassadorTick and the multi threaded version includes a separate thread: esimRTIthread. The function (and the thread) replace the tick method of the RTI::RTIambassador.
An overview of the differences is presented in the following table (in the table tick should be read as esimRTIrtiAmbassadorTick):

Mode	**Single Threaded Direct**	**Single Threaded Buffered**	**Multi Threaded Direct**	**Multi Threaded Buffered**
Buffer RTI calls	NO: the RTI calls are executed directly	YES	NO: the RTI calls are executed directly	YES
Buffer RTI callbacks	YES	YES	YES	YES
Execute the RTI calls	the RTI calls are executed directly	Tick reads and executes one call from the RTI call buffer	The RTI calls are executed directly	esimRTIthread reads and executes all calls from the RTI call buffer
The RTI is ticked by:	tick	tick	tick	esimRTIthread
Buffer the RTI callbacks	YES	YES	YES	YES
Execute the RTI callbacks	Tick reads and executes all callbacks from the RTI callback buffer	Tick reads and executes one callback from the RTI callback buffer	Tick reads and executes all callbacks from the RTI callback buffer	Tick reads and executes one callback from the RTI callback buffer

Table I.1: Overview of the functionality of esimRTIrtiAmbassadorTick function in the available combinations of EsimRTI libraries (single-threaded vs. multi-threaded) and EsimRTI mode of operation (direct mode vs. buffered mode). In the table tick should be read as esimRTIrtiAmbassadorTick.

The esimRTIrtiAmbassadorTick should be called on a regular basis. The frequency depends on the number of RTI calls to and RTI callbacks from the federation. The esimRTIrtiAmbassadorTick should be called from a non-real-time task.
The EsimRTI can be queried for the actual number of pending RTI calls and RTI callbacks with the following functions:

- esimRTIrtiAmbassadorGetNbufferCalls, and

- esimRTIfederateAmbassadorGetNbufferCallbacks.

I.3 Porting an existing federate to EsimRTI

An existing federate can be ported to EsimRTI mainly by making some syntactical replacements and some extra coding.

I.3.1 Type Conversions

The following table shows how the RTI types and classes are mapped onto their C equivalents.

RTI/C++ types/classes	C typedefs
RTI::FedTime	EsimRTIfedTime
RTI::FedTime&	EsimRTIfedTime *
RTI::AttributeHandle RTI::ParameterHandle RTI::ObjectHandle RTI::ObjectClassHandle RTI::InteractionClassHandle RTI::FederateHandle	EsimRTIattributeHandle EsimRTIparameterHandle EsimRTIobjectHandle EsimRTIclassHandle EsimRTIinteractionHandle EsimRTIfederateHandle
RTI::RTIambassador	EsimRTIrtiAmbassadorHandle
RTI::FederateAmbassador	EsimRTIfederateAmbassadorHandle
RTI::...HandleSet	EsimRTIattributeHandleSet EsimRTIfederateHandleSet EsimRTIhandleSet EsimRTIparameterHandleSet
RTI::...HandleValuePairSet	EsimRTIattributeHandleValuePairSet EsimRTIhandleValuePairSet EsimRTIparameterHandleValuePairSet
RTI::EventRetractionHandle	ErsimRTIeventRetractionHandle
RTI::Boolean	EsimRTIboolean

Table I.2: Mapping of RTI types and classes onto equivalent C types.

I.3.2 Constructors and destructors

The following table shows how the RTI constructors and destructors are mapped onto C equivalent functions.

C++ constructor / destructor	C function
RTI::AttributeHandleSetFactory.create	esimRTIattributeHandleSetNew
delete RTI::AttributeHandleSet	esimRTIattributeHandleSetDelete
RTI::AttributeSetFactory.create	esimRTIattributeHandleValuePairSetNew
delete RTI::AttributeHandleValuePairSet	esimRTIattributeHandleValuePairSetDelete

Table I.3: Mapping of C++ constructors and destructors onto equivalent C functions.

C++ constructor / destructor	C function
new RTI::FederateAmbassador	esimRTIfederateAmbassadorNew
delete RTI::FederateAmbassador	esimRTIfederateAmbassdadorDelete
RTI::FederateHandleSetFactory.create	esimRTIfederateHandleSetNew
delete RTI::FederateHandleSet	esimRTIfederateHandleSetDelete
RTI::ParameterSetFactory.create	esimRTIparameterHandleValuePairSetNew
delete RTI::ParameterHandleValuePairSet	esimRTIparameterHandleValuePairSetDelete
new RTI::RTIambassador	esimRTIrtiAmbassadorNew
delete RTI::RTIambassador	esimRTIrtiAmbassdadorDelete

Table I.3: Mapping of C++ constructors and destructors onto equivalent C functions.

I.3.3 Method naming convention

The RTI::RTIambassador methods are replaced with functions of which the name is formed by the combination of class and method. In general Class.method should be replaced with esimRTI<class><Method>.

C++ method	C function
RTI::FederateAmbassador.method	esimRIfederateAmbassadorMethod
RTI::RTIambassador.method	esimRTIrtiAmbassadorMethod

Table I.4: Mapping of Class.methods onto esimRTIclassMethods.

I.3.4 Useful Constants

A few useful C constants to use with the EsimRTI are:

Type	Constant
EsimRTIfedTime	esimRTIinifiniteTime
EsimRTIfedTime	esimRTIepsilonTime
EsimRTIeventRetractionHandle	esimRTIzeroRetractionHandle

Table I.5: C constants.

I.3.5 Syntactical replacements

All parameters by reference (RTI::FedTime& time) have been replaced by pointer types (EsimRTIfedTime* time).

I.3.6 Exceptions

Exceptions thrown by the RTI are caught, and an esimMessage is displayed, providing the user with the reason of the error, the calling function and the exception name. Also, when appropriate, 0 or NULL is returned.
The format of the esimMessage is:

```
ERROR: Exception <exception name> thrown in <functionName ()>,
reason: <reason of the exception>
```

The corresponding (possibly buffered) RTI function call is not executed.
The display of Exception messages can be switched off by calling esimRTItoggleExceptionMessages().
An example of the above is the EsimRTI-version of esimRTIrtiAmbassadorJoinFederationExecution: it does not throw the exceptions like the RTI equivalent does. This creates the following situation: the application cannot sense the exception, and in particular, the exception FederationExecutionDoesNotExist. The best way to check whether the join succeeded, is to check whether esimRTIrtiAmbassadorHasJoined returns EsimRTItrue. An alternative implementation is to check whether the federateId returned is not equal to NULL. The preferred implementation is:

```
numTries = 0;

while (esimRTIrtiambassadorHasJoined () == EsimRTIfalse && numTries++ < 20)
     {
  federateId = esimRTIrtiAmbassasdorJoinFederationExecution(
    federateName, federationName, &esimFederateAmbassador);
  tick();
  esimRTImilliSleep(1000); // sleep 1 second
}

if (esimRTIambassadorHasJoined() == EsimRTIfalse) {
  exit();
}
```

I.3.7 Joining a federation execution

A federation execution is joined by calling the method esimRTIrtiAmbassadorJoinFederationExecution until the method esimRTIrtiAmbassadorHasJoined returns EsimRTItrue.

I.3.8 Registering an object Instance

The EsimRTI implementation of RTIambassador::registerObjectInstance is called esimRTIrtiAmbassadorRegisterObjectInstance. Depending on the current mode of operation of the EsimRTI: buffer or direct the function differs in its behavior:

- EsimRTI mode is buffered
- The esimRTIrtiAmbassadorRegisterObjectInstance is asynchronous and returns 0 as the EsimRTIobjectHandle return value. The EsimRTIobjectHandle is delivered to the federate through the following EsimRTI specific callback:
- void esimRTIfederateAmbassadorObjectInstanceRegistrationSucceeded (EsimRTIclassHandle theClass, EsimRTIobjectHandle theObject, const char* theName)
- EsimRTI mode is direct
- The esimRTIrtiAmbassadorRegisterObjectInstance functions as the original RTI call would. However the method calls the method (to be implemented for the federate) esimRTIfederateAmbassadorRegisterObjectInstanceSucceeded as well, to allow the federate to link the RTI object handle to the application object.

Linking the registration of object handles with the appropriate names through this callback can be done in two ways:

- register only one object of a certain class at a time, wait for the callback to be delivered, and then register the next object of the class;
- use the object name to link the callback to the registration;

The file esimRTIobjects.h contains simple structures to register the linkage of classes, objects, interactions, attributes and parameters in the application (C-domain) to the RTI domain (C++). These structures are described in the next chapter.

I.3.9 Callback functions

The RTI federate ambassador methods are replaced with federate ambassador callback functions to be implemented for the application (the federate). Each of the functions listed below needs to be implemented by the developer that links the EsimRTI-library into its application.
A template implementation of these functions is provided in the source file $EFOROOT/Examples/Flywheel/esimRTIfederateAmbassadorCallbacks.c. A non-void implementation of a callback can be created by filling the body of the function in a renamed copy of this file.

I.3.9.1 Overview of federate ambassador callbacks that require implementation

EsimRTI specific callback

esimRTIfederateAmbassadorObjectInstanceRegistrationSucceeded

Federation management callbacks

esimRTIfederateAmbassadorSynchronizationPointRegistrationSucceeded
esimRTIfederateAmbassadorSynchronizationPointRegistrationFailed
esimRTIfederateAmbassadorAnnounceSynchronizationPoint
esimRTIfederateAmbassadorFederationSynchronized
esimRTIfederateAmbassadorInitiateFederateSave
esimRTIfederateAmbassadorFederationSaved
esimRTIfederateAmbassadorFederationNotSaved
esimRTIfederateAmbassadorRequestFederationRestoreSucceeded
esimRTIfederateAmbassadorRequestFederationRestoreFailed
esimRTIfederateAmbassadorFederationRestoreBegun
esimRTIfederateAmbassadorInitiateFederateRestore
esimRTIfederateAmbassadorFederationRestored
esimRTIfederateAmbassadorFederationNotRestored

Declaration management callbacks

esimRTIfederateAmbassadorStartRegistrationForObjectClass
esimRTIfederateAmbassadorStopRegistrationForObjectClass
esimRTIfederateAmbassadorTurnInteractionsOn
esimRTIfederateAmbassadorTurnInteractionsOff

Object management callbacks

esimRTIfederateAmbassadorDiscoverObjectInstance
esimRTIfederateAmbassadorReflectAttributeValues
esimRTIfederateAmbassadorReceiveInteraction
esimRTIfederateAmbassadorRemoveObjectInstance
esimRTIfederateAmbassadorProvideAttributeValueUpdate
esimRTIfederateAmbassadorTurnUpdatesOnForObjectInstance
esimRTIfederateAmbassadorTurnUpdatesOffForObjectInstance

Ownership management callbacks

esimRTIfederateAmbassadorRequestAttributeOwnershipAssumption
esimRTIfederateAmbassadorAttributeOwnershipDivestitureNotification
esimRTIfederateAmbassadorAttributeOwnershipAcquisitionNotification
esimRTIfederateAmbassadorAttributeOwnershipUnavailable
esimRTIfederateAmbassadorRequestAttributeOwnershipRelease
esimRTIfederateAmbassadorConfirmAttributeOwnershipAcquisitionCancellation
esimRTIfederateAmbassadorInformAttributeOwnership
esimRTIfederateAmbassadorAttributeIsNotOwned
esimRTIfederateAmbassadorAttributeOwnedByRTI

Time management callbacks

esimRTIfederateAmbassadorTimeRegulationEnabled
esimRTIfederateAmbassadorTimeConstrainedEnabled

esimRTIfederateAmbassadorTimeAdvanceGrant
esimRTIfederateAmbassadorRequestRetraction

I.3.10 Get names and get handles

The ancillary functions that return a value esimRTIrtiAmbassadorGet...Name and esimRTIrtiAmbassadorGet...Handle can only be called in Direct mode. In Buffered mode, a warning is being displayed, and the function returns NULL or 0 respectively.
This poses no problem, since:

- object names are delivered by means of the following callbacks: esimRTIfederateAmbassadorObjectInstanceRegistrationSucceeded and esimRTIfederateAmbassadorDiscoverObjectInstance;
- the other functions are needed during initialization only, which normally occurs in the Direct mode

I.3.11 Time queries

The RTI Time management methods can be called in both the Direct and Buffered modes:

RTI / C++ domain	**C domain**
queryFederateTime	esimRTIrtiAmbassadorQueryFederateTime
queryLBTS	esimRTIrtiAmbassadorQueryLBTS
queryFederateTime	esimRTIrtiAmbassadorQueryFederateTime
queryMinNextEventTime	esimRTIrtiAmbassadorQueryMinNextEventTime
queryLookahead	esimRTIrtiAmbassadorQueryLookahead
	esimRTIrtiAmbassadorQueryTimes

Table I.6: Mapping of RTI query time methods onto C functions.

In the Direct mode the functions call the RTI equivalents directly. In the Buffered mode these functions return relevant values as they have been copied from the RTI when the EsimRTI tick function is called.

I.3.12 Function overloading

The RTI uses function overloading to differentiate between different implementations of a function. Fortunately, when a function is being overloaded in the RTI, there are always at most 2 versions of the function, and the set of parameters belonging to the one is a subset of the parameters (least informative version) belonging to other (most informative version). Since C does not support overloading, the most informative functions have been implemented in the EsimRTI.
Choosing between the invocations of a certain RTI method would normally be achieved by means of the distinct lists of parameters. In the EsimRTI the least informative functions are available by means of calling the most informative ones with the non-relevant parameters set to 0 or NULL.

I.3.12.1 Examples that illustrate how functions are mapped onto overloaded methods and vice versa.

The function call:

```
esimRTIrtiAmbassadorRegisterObjectInstance (
     theClassHandle, NULL)
```

will invoke the method:

```
RTI::RTIambassador::registerObjectInstance (
     theClassHandle)
```

The function call:

```
esimRTIrtiAmbassadorRegisterObjectInstance (
     theClassHandle, objectName)
```

will invoke the method:

```
RTI::RTIambassador::registerObjectInstance (
     theClassHandle, objectName)
```

The method:

```
RTI::FederateAmbassadorassador::removeObjectInstance (
     objectHandle, time, tag, retractionHandle)
```

will call the function:

```
esimRTIfederateAmbassadorRemoveObjectInstance (
     objectHandle, time, tag, retractionHandle)
```

The method:

```
RTI::FederateAmbassadorassador::removeObjectInstance (
     objectHandle, tag)
```

will call the function:

```
esimRTIfederateAmbassadorRemoveObjectInstance (
     objectHandle, NULL, tag, esimRTIzeroRetractionHandle)
```

I.3.13 EsimRTI mode change and mode dependent functions

The EsimRTI mode is can be changed from direct to buffered using the esimRTImodeSet function.
A number of functions are only supported if the EsimRTI mode is direct (e.g. before esimRTImodeSet (EsimRTImodeBuffered) or after esimRTImodeSet (EsimRTImodeDirect) is called). In general these functions are related to initialization and finalization. This means that the initialization and the finalization of the federate have to be done in Direct mode. For most applications and simulations this is no problem, except for the function getObjectInstanceHandle and getObjectInstanceName, which will typically be needed throughout the simulation.
If one of the direct mode only functions mentioned below is called in buffered-mode, the following message will be printed on the console:

```
 WARNING: Cannot invoke <method> in Buffered mode
```

and the function will return 0 or NULL.

Name	**Initialization**	**Finalization**
esimRTIfederateAmbassadorNew	I	
esimRTIfederateAmbassadorDelete		F
esimRTIrtiAmbassadorNew	I	
esimRTIrtiAmbassadorDelete		F[1]
esimRTIrtiAmbassadorCreateFederationExecution	I	
esimRTIrtiAmbassadorDestroyFederationExecution		F[2]
esimRTIrtiAmbassadorResignFederationExecution		F[1]
esimRTIrtiAmbassadorJoinFederationExecution	I	

Table I.7: Overview of functions available in direct mode only.

Name	**Initialization**	**Finalization**
esimRTIrtiAmbassadorGetAttributeName	I	
esimRTIrtiAmbassadorGetAttributeHandle	I	
esimRTIrtiAmbassadorGetInteractionClassHandle	I	
esimRTIrtiAmbassadorGetInteractionClassName	I	
esimRTIrtiAmbassadorGetObjectClass	I	
esimRTIrtiAmbassadorGetObjectClassHandle	I	
esimRTIrtiAmbassadorGetObjectClassName	I	
esimRTIrtiAmbassadorGetObjectInstanceHandle	I	
esimRTIrtiAmbassadorGetObjectInstanceName	I	
esimRTIrtiAmbassadorGetOrderingHandle	I	
esimRTIrtiAmbassadorGetOrderingName	I	
esimRTIrtiAmbassadorGetParameterHandle	I	
esimRTIrtiAmbassadorGetParameterName	I	
esimRTIrtiAmbassadorGetTransportationHandle	I	
esimRTIrtiAmbassadorGetTransportationName	I	

Table I.7: Overview of functions available in direct mode only.

I.3.14 Memory Management

All memory that is referenced in the real-time domain should be allocated by means of the real-time memory allocator esimMalloc and its derivatives esimCalloc, esimRealloc and esimStrdup. It should be freed by esimFree.
The EsimRTI library overrides the operators new and delete for all of its classes. The global new and delete are not overridden, since this gives some problems on IRIX-platform.
Memory conventions of passing arguments in the EsimRTI are exactly similar to the RTI-conventions. The following table lists the 6 different ways of passing parameters as defined by the RTI.

C1	In parameter by value.
C2	Out parameter by reference.
C3	Function return by value.
C4	In parameter by const reference. Caller provides memory. Caller may free memory or overwrite it upon completion of the call. Callee must copy during the call anything it wishes to save beyond completion of the call. Parameter type must define const accessor methods.
C5	Out parameter by reference. Caller provides reference to object. Callee constructs an instance on the heap (new) and returns. The caller destroys the instance (delete) at its leisure.
C6	Function return by reference. Callee constructs an instance on the heap (new) and returns a reference. The caller destroys the instance (delete) at its leisure.

Table I.8: Different ways of passing arguments as defined by the RTI.

[1] These methods automatically change the EsimRTI mode to Direct.

[2] This method returns EsimRTIfalse if the federation execution was not destroyed (because other federates still have joined the federation execution). EsimRTItrue is returned if there is an RTI internal error or if the federation execution does not exist. Exception warning are printed through esimWarning in all cases.

I.3.15 Limitations and performance

The buffer size of the Federate callback buffer and the RTI ambassador call buffer is defined as 256 currently. Regular calls of the EsimRTIrtiAmbassadorTick are required to flush both buffers.
The extra performance degradation caused by the EsimRTI (when compared to the RTI) is minimal. Delays caused by the buffering can be minimized by calling the EsimRTIrtiAmbassadorTick tick at an appropriate frequency.

I.3.16 Missing Functionality

All functionality regarding regions, spaces, data distribution management, declaration management handling data distribution management, extents, and dimensions is not implemented in EsimRTI.

I.4 EsimRTIobject and EsimRTIvariable structures

The EsimRTIobject and EsimRTIvariable data structures are the basis for the development of facilities that can be used to ease the association of application data with HLA/RTI classes, objects, interactions, attributes and parameters.
The EsimRTIobject and EsimRTIvariable data structures do not have equivalents in the HLA/RTI specification.

I.4.1 EsimRTIobject

I.4.1.1 Structure

```
typedef struct EsimRTIobject
{
  char *              className;
  EsimRTIclassHandle  classHandle;
  char *              objectName;
  EsimRTIobjectHandle objectHandle;
  EsimRTIobjectType   esimRTIobjectType;
  EsimRTIpublishType  esimRTIpublishType;
} EsimRTIobject, *EsimRTIobjectRef;
```

I.4.1.2 Methods

```
EsimRTIobject *esimRTIobjectNew(
  char *              theClassName,
  EsimRTIclassHandle  theClassHandle,
  char *              theObjectName,
  EsimRTIobjectHandle theObjectHandle,
  EsimRTIobjectType   theObjectType,
  EsimRTIpublishType  thePublishType);
```

Allocates memory for a new EsimRTIobject and initializes the fields. The char * arguments are copied to new allocated memory. The method uses esimMalloc.

```
EsimRTIobject *dsimRTIobjectDelete(
  EsimRTIobject *theObject);
```

Frees the memory occupied by the given EsimRTIobject (including the char * fields referred to). The method uses esimFree.

I.4.2 EsimRTIvariable

I.4.2.1 Structure

```
typedef struct
{
  char *              variableName;
  EsimRTIhandle       variableHandle;
  EsimRTIvariableType variableType;
  void *              variableRef;
  unsigned int        nArrayEntries;
  char *              objectName;
  EsimRTIobject *     pEsimRTIobject;
} EsimRTIvariable, *EsimRTIvariableRef;
```

Notes:

- The nArrayEntries field is only applicable if the variableType field represents one of the array types.
- Note that the variable reference in the EsimRTIvariable struct always refers to the actual variable (contains the address of the variable and not of the buffer with the encoded value).
- The objectName field looks redundant but it is not: the EsimRTIvariable structure can be used to define a series of variables (e.g. using a const array in the model source code) where the address of the corresponding EsimRTIobject is not known or complex to retrieve: the objectName can be used to find the right EsimRTIobject.

I.4.2.2 Methods

```
EsimRTIvariable *esimRTIvariableNew (
  char *              theVariableName,
  EsimRTIhandle       theVariableHandle,
  EsimRTIvariableType theVariableType,
  void *              theVariableRef,
  char *              theObjectName,
  EsimRTIobject *     theEsimRTIobject);
```

The method uses createEsimRTIvariableArrayNew (with 0 as theNarrayEntries argument).

```
EsimRTIvariable *esimRTIvariableArrayNew(
  char *              theVariableName,
  EsimRTIhandle       theVariableHandle,
  EsimRTIvariableType theVariableType,
  void *              theVariableRef,
  unsigned int        theNarrayEntries,
  char *              theObjectName,
  EsimRTIobject *     theEsimRTIobject);
```

Allocates memory for a new EsimRTIvariable and initializes the fields. The char * arguments are copied to new allocated memory. The method uses esimMalloc. Because the method is used by esimRTIvariableNew the method does not check whether theVariableType is one of the array types.

```
EsimRTIvariable *esimRTIvariableDelete(
  EsimRTIvariable * theVariable);
```

Frees the memory occupied by the given EsimRTIvariable (including the char * fields referred to). The method uses esimFree.

I.4.2.3 Enumerated types

The following table shows the supported values for the enumerated types used in the previous sections: EsimRTIobjectType, EsimRTIpublishType and EsimRTIvariableType.

Enumerated type	Enumerated values		
EsimRTIobjectType	class		
	object		
	interaction		
EsimRTIpublishType	subscribe		
	publish		
	publish-subscribe		
EsimRTIvariableType	char	-	-
	short	short-string	short-array
	unsigned-short	unsigned-short-string	unsigned-short-array
	int	int-string	int-array
	unsigned-int	unsigned-int-string	unsigned-int-array
	float	float-string	float-array
	double	double-string	double-array
	string	-	-

Table I.9: EsimRTIobject and EsimRTIvariable related enumerated types.

Notes:

- The ...-string in the enumerated types represents that the variable is encoded as a string using appropriate %c, %ld and %lf sprintf and sscanf format specifiers.
- The other enumerated types are encoded as is: as a series of bytes as expected on the current platform. Note that this is valid for data transfer on single platform type federations only.
- The -array in the enumerated types represents that the array will be encoded as is: as a series of bytes as expected on the current platform. Note that this is valid use for single platform federations only. The array dimensions are supposed to be well known (are defined in the SOM of the federate).

I.5 HandleValuePairSet encoding and decoding facilities

The EsimRTI has been extended with basic facilities that ease the encoding and decoding of attributes and parameters. These EsimRTIhandleValuePairSet facilities do not have equivalents in the HLA/RTI specification. The supported facilities are available for the EsimRTI equivalents of the RTI classes AttributeHandleValuePairSet and ParameterHandleValuePairSet.
The following sections contain the prototypes of the encoding and decoding functions for EsimRTIhandleValuePairSet. The following table shows how the names of these functions can be used to determine the function names for the EsimRTIattributeHandleValuePairSets and EsimRTIparameterHandleValuePairSets.

Replace:	**With:**
esimRTIhandleValuePairSet...	esimRTIattributeHandleValuePairSet... or esimRTIparameterHandleValuePairSet...
EsimRTIhandle	EsimRTIattributeHandle or EsimRTIparameterHandle
EsimRTIhandleValuePairSet	EsimRTIattributeHandleValuePairSet or EsimRTIparameterHandleValuePairSet

Table I.10: Naming conventions for the attribute and parameter encoding and decoding functions and types listed in the following sections.

The list of encoding and decoding facilities can easily be extended if the need arises.

I.5.1 Encoding methods

```
void esimRTIhandleValuePairSetAddDouble(
  EsimRTIhandle               handle,
  double                      number,
  EsimRTIhandleValuePairSet * theSet);

void esimRTIhandleValuePairSetAddDoubleAsString(
  EsimRTIhandle               handle,
  double                      number,
  EsimRTIhandleValuePairSet * theSet);

void esimRTIhandleValuePairSetAddLong(
  EsimRTIhandle               handle,
  long                        number,
  EsimRTIhandleValuePairSet * theSet);

void esimRTIhandleValuePairSetAddLongAsString(
  EsimRTIhandle               handle,
  long                        number,
  EsimRTIhandleValuePairSet * theSet);

void esimRTIhandleValuePairSetAddString(
  EsimRTIhandle               handle,
  const char *                string,
  EsimRTIhandleValuePairSet * theSet);
```

I.5.2 Decoding methods

```
double esimRTIhandleValuePairSetGetDouble(
  unsigned long                     index,
  const EsimRTIhandleValuePairSet * theSet);

double esimRTIhandleValuePairSetGetDoubleFromString(
  unsigned long                     index,
  const EsimRTIhandleValuePairSet * theSet);

long getLongFromHandleValuePairSet(
  unsigned long                     index,
  const EsimRTIhandleValuePairSet * theSet);
```

```
long esimRTIhandleValuePairSetGetLongFromString(
  unsigned long                    index,
  const EsimRTIhandleValuePairSet * theSet);

char * esimRTIhandleValuePairSetCopyString(
  unsigned long                    index,
  const EsimRTIhandleValuePairSet * theSet);
```

I.6 Classes, structures and header files of the EsimRTI

This chapter contains a list and short description of the classes (C++), structs (C) and the header files that implement the EsimRTI library.

I.6.1 Classes

The following table shows the classes that are used within the EsimRTI to support the RTI functionality.

EsimRTI type or class	**Description**
EsimRTIbooleanClass	Equivalent of RTI::Boolean
EsimRTIbufferMethod	Base class for entries (calls and callbacks) of the method buffers (RTI calls and Federate callbacks)
EsimRTImethodBuffer	Base class for ring buffers to store methods (RTI calls and Federate callbacks)
EsimRTIfederateAmbassadorCallbackBuffer	The Federate ambassador callback buffer
EsimRTIrtiAmbassadorCallBuffer	The RTI ambassador call buffer
EsimRTIeventRetractionHandleClass	Equivalent of RTI::EventRetractionHandle
EsimRTIfedTimeClass	Container for all federate time attributes
EsimRTIfederateAmbassador	Extends RTI::FederateAmbassador
EsimRTIfederateAmbassadorCallback	Base class for Federate ambassador calls to be inserted in the Federate ambassador callback buffer
...Callback	The Federate ambassador callbacks to be inserted in the Federate ambassador callback buffer
EsimRTIhandleSetClass	Equivalent of RTI::HandleSet
EsimRTIhandleValuePairSetClass	Equivalent of RTI::HandleValuePairSet
EsimRTIrtiAmbassador	Extends RTI::RTIambassador
EsimRTIrtiAmbassadorCall	Base class for RTI ambassador calls to be inserted in the RTI ambassador call buffer
...Call	The RTI ambassador calls to be inserted in the RTI ambassador call buffer

Table I.11: EsimRTI types and classes.

I.6.2 Structures

The following tables shows the structures that are used to access the EsimRTI functions equivalent with the RTI methods.

RTI / C++ domain	**C domain**
RTI::EventRetraction structure	EsimRTIeventRetractionHandle
RTI::HandleSet	EsimRTIhandleSet
RTI::AttributeHandleSet	EsimRTIattributeHandleSet
RTI::FederateHandleSet	EsimRTIfederateHandleSet
-	EsimRTIvaluePair
RTI::ParameterHandleSet	EsimRTIparameterHandleSet
RTI::AttributeHandleValuePairSet	EsimRTIattributeHandleValuePairSet
RTI::ParameterHandleValuePairSet	EsimRTIparameterHandleValuePairSet
-	EsimRTItimeTemplate
RTI::RTIambassador class	EsimRTIrtiAmbassador
RTI::FederateAmbassador class	EsimRTIfederateAmbassador
-	EsimRTIobject[3] Structure that can be used to associate application objects to HLA/RTI objects
-	EsimRTIvariable[3] Structure that can be used to associate application variables to HLA/RTI parameters and attributes

Table I.12: RTI/ C++ types and classes mapped onto equivalent C structures.

I.6.3 Header files

The developer using the EsimRTI should include only the esimRTI.h header file. This esimRTI.h header file itself includes a number of headers, just as the RTI.hh-header of the RTI does.
An optional second file to include is esimRTIshortnames.h that defines short cuts for the (sometimes very) long names of the EsimRTI.
The following table lists the header files of the EsimRTI, the RTI equivalent and a short description.

EsimRTI	**RTI equivalent**	**Contents**
esimRTI.h	RTI.hh	Basic header
esimRTIboolean.h	baseTypes.hh	Boolean conversions between C and RTI
esimRTIeventRetractionHandle.h	RTItypes.hh	EventRetractionHandle conversions between C and RTI
esimRTIfedTime.h	fedTime.hh	Time conversions between C and RTI
esimRTIfederateAmbassador.h	RTI.hh federateAmbServices.hh NULLfederateAmbSer-vives.hh	Federate Ambassador

Table I.13: EsimRTI header files, RTI equivalents and short descriptions.

[3]These structures can be used to register information of object classes, attributes, interactions and parameters. The registered information can be used to associate application variables to their RTI equivalents. Note that these structures are not part of the HLA/RTI specification.

EsimRTI	**RTI equivalent**	**Contents**
esimRTIfederateAmbassadorCallbacks.h	federateAmbServices.hh	Federate Ambassador methods (callbacks)
esimRTIhandleSet.h	RTItypes.hh	HandleSet conversions between EsimRTI and RTI
esimRTIhandleValuePairSet.h	RTItypes.hh	HandleValueSet conversions between EsimRTI and RTI
esimRTImutex.h	-	Mutual exclusion functions for the EsimRTI
esimRTIobjects.h	-	Data structures and enumerated types that can be used associate application objects and variables to HLA/RTI classes, objects, interactions, attributes and parameters
esimRTIrtiAmbassador.h	RTI.hh RTIambServices.hh	RTI Ambassador
esimRTIrtiAmbassadorCalls.h	RTIambServices.hh	RTI Ambassador functions (calls)
esimRTIshortNames.h	-	Short equivalents for the long EsimRTI names
esimRTIstandalone.h	-	Functions required for the stand-alone versions of the EsimRTI library
esimRTIstring.h	-	String manipulation functions
esimRTIthread.h	-	esimRTIthread and related functions for the multi-threaded EsimRTI library
esimRTItypes.h	baseTypes.hh	Basic typedefs and classes

Table I.13: EsimRTI header files, RTI equivalents and short descriptions.

I.7 Flywheel example

With the development of the EsimRTI a new example has been added to the EuroSim source code management repository: $EFOROOT/Examples/Flywheel. This new example is based on the Satellite example and consists of an engine model and a flywheel model.
An engine keeps a flywheel spinning with a RPM that lies within certain limits. The engine is started if the flywheel RPM is below the minimum RPM and is turned off if the flywheel RPM is above the maximum RPM. Note that the purpose of the example is to illustrate the implementation of the EsimRTI and that neither the model of the engine nor the model of the flywheel are physically correct.
The Flywheel example can be run in three different modes of operation:

- as one integrated model within EuroSim (flywheelengine.model) using one Simulation Controller.

This version does not use HLA/RTI for the data communication between engine and flywheel.

- as two models within EuroSim using two Simulation Controllers (the engine.model and the flywheel.model). This version uses the EsimRTI library to exchange data through the RTI between engine and flywheel.
- as two stand-alone programs (build using Makefile) without EuroSim. This version uses the EsimRTI library to exchange data through the RTI between engine and flywheel.

I.8 RTI API compared with the EsimRTI API

Table I.14 maps the RTI methods onto their EsimRTI equivalent functions.
The table also includes the short names as defined in esimRTIshortNames.h. That file includes a short name if the EsimRTI API cell contains two entries.

RTI API	**EsimRTI API**	**Remark**
-	esimRTI<>HandleSetCopy	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetAddDoubleAsString	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetAddLongAsString	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetCopy	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetCopyString	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetGetDouble	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetGetDoubleFromString	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetGetLong	Available for Attribute and Parameter sets
-	esimRTI<>HandleValuePairSetGetLongFromString	Available for Attribute and Parameter sets
-	esimRTIepsilonTime	10e-9 default lookahead
-	esimRTIeventRetractionHandleDelete	Frees EsimRTIeventRetractionHandle
-	esimRTIfederateAmbassadorGetHandle	Returns esimRTIfederateAmbassador
-	esimRTIfederateAmbassadorGetNbufferCallbacks	Returns # pending FederateAmbassador callbacks
-	esimRTIfederateAmbassadorObjectInstanceRegistrationSucceeded objectInstanceRegistrationSucceeded	
-	esimRTIfedTimeDelete	Frees EsimRTIfedTime

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	**EsimRTI API**	**Remark**
-	esimRTImodeGet	Returns EsimRTImodeDirect or EsimRTImodeBuffered
-	esimRTImodeSet	Set mode to EsimRTImodeDirect or EsimRTImodeBuffered
-	esimRTIobjectDelete	Frees EsiRTIobject
-	esimRTIobjectNew	Allocates EsiRTIobject
-	esimRTIpositiveInfiniteTime	Upperbound for federation logical time axis
-	esimRTIrtiAmbassadorGetHandle	Returns the EsimRTIrtiAmbassador
-	esimRTrtiAambassador.GetNbufferCalls	Returns # pending RTIambassador calls
-	esimRTIrtiAmbassadorIsJoined	Returns EsimRTItrue if the federate has joined the federation
-	esimRTIrtiAmbassadorQueryTimes	Query all RTIambassdor times
-	esimRTItoggleExceptionMessages	Show/hide exceptions
-	esimRTIvariableArrayNew	Allocates array of EsimRTIvariables
-	esimRTIvariableDelete	Frees EsimRTIvariable
-	esimRTIvariableNew	Allocates EsimRTIvariable
-	esimRTIzeroRetractionHandle	{0, 0} retraction handle
˜RTI::<>HandleSet	esimRTI<>HandleSetDelete	Available for Attribute, Federate and Parameter sets
˜RTI::<>HandleValuePairSet	esimRTI<>HandleValuePairSetDelete	Available for Attribute and Parameter sets
˜RTI::FederateAmbassador	esimRTIfederateAmbassadorDelete	Destroys the EsimRTIfederateAmbassador
˜RTI::RTIambassador	esimRTIrtiAmbassadorDelete	Destroys the EsimRTIrtiAmbassador

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	EsimRTI API	Remark
new RTI::FederateAmbassador	esimRTIfederateAmbassadorNew	Creates the EsimRTIfederateAmbassador
new RTI::RTIambassador	esimRTIrtiAmbassadorNew	Creates the EsimRTIrtiAmbassador
RTI::<>HandleSet.add	esimRTI<>HandleSetAdd	Available for Attribute, Federate and Parameter Sets
RTI::<>HandleSet.empty	esimRTI<>HandleSetEmpty	Available for Attribute, Federate and Parameter sets
RTI::<>HandleSet.getHandle	esimRTI<>HandleSetGetHandle	Available for Attribute, Federate and Parameter sets
RTI::<>HandleSet.isEmpty	esimRTI<>HandleSetIsEmpty	Available for Attribute, Federate and Parameter sets
RTI::<>HandleSet.isMember	esimRTI<>HandeSetHandleIsMember	Available for Attribute, Federate and Parameter sets
RTI::<>HandleSet.remove	esimRTI<>HandleSetRemove	Available for Attribute, Federate and Parameter sets
RTI::<>HandleSet.size	esimRTI<>HandleSetSize	Available for Attribute, Federate and Parameter sets
RTI::<>HandleSetFactory	esimRTI<>HandleSetNew	Available for Attribute, Federate and Parameter sets
RTI::<>HandleValuePairSet.add	esimRTI<>HandleValuePairSetAdd	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.empty	esimRTI<>HandleValuePairSetEmpty	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.getHandle	esimRTI<>HandleValuePairSetGetHandle	Available for Attribute and Parameter sets

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	EsimRTI API	Remark
RTI::<>HandleValuePairSet.getTransportType	esimRTI<>HandleValuePairSetGetTransportType	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.getValue	esimRTI<>HandleValuePairSetGetValue	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.getValueLength	esimRTI<>HandleValuePairSetGetValueLength	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.getValuePointer	esimRTI<>HandleValuePairSetGetValuePointer	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.moveFrom	esimRTI<>HandleValuePairSetMoveFrom	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.remove	esimRTI<>HandleValuePairSetRemove	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSet.size	esimRTI<>HandleValuePairSetGetSize	Available for Attribute and Parameter sets
RTI::<>HandleValuePairSetFactory	esimRTI<>HandleValuePairSetNew	Available for Attribute and Parameter sets
RTI::AttributeHandleValuePairtSet<>	-	See RTI::<>HandleValuePairtSet<>
RTI::AttrinbuteHandleSet<>	-	See RTI::<>HandleSet<>
RTI::FederateAmbassador.announceSynchronizationPoint	esimRTIfederateAmbassadorAnnounceSynchronizationPoint announceSynchronizationPoint	
RTI::FederateAmbassador.attributeIsNotOwned	esimRTIfederateAmbassadorAttributeIsNotOwned attributeIsNotOwned	
RTI::FederateAmbassador.attributeOwnedByRTI	esimRTIfederateAmbassadorAttributeOwnedByRTI attributeOwnedByRTI	
RTI::FederateAmbassador.attributeOwnershipAcquisitionNotification	esimRTIfederateAmbassadorAttributeOwnershipAcquisitionNotification attributeOwnershipAcquisitionNotification	
RTI::FederateAmbassador.attributeOwnershipDivestitureNotification	esimRTIfederateAmbassadorAttributeOwnershipDivestitureNotification attributeOwnershipDivestitureNotification	

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	**EsimRTI API**	**Remark**
RTI::FederateAmbassador.attributeOwnershipUnavailable	esimRTIfederateAmbassadorAttributeOwnershipUnavailable attributeOwnershipUnavailable	
RTI::FederateAmbassador.confirmAttributeOwnershipAcquisitionCancellation	esimRTIfederateAmbassadorConfirmAttributeOwnershipAcquisitionCancellation attributeOwnershipAcquisitionCancellation	
RTI::FederateAmbassador.discoverObjectInstance	esimRTIfederateAmbassadorDiscoverObjectInstance discoverObjectInstance	
RTI::FederateAmbassador.federationNotRestored	esimRTIfederateAmbassadorFederationNotRestored federationNotRestored	
RTI::FederateAmbassador.federationNotSaved	esimRTIfederateAmbassadorFederationNotSaved federationNotSaved	
RTI::FederateAmbassador.federationRestoreBegun	esimRTIfederateAmbassadorFederationRestoreBegun federationRestoreBegun	
RTI::FederateAmbassador.federationRestored	esimRTIfederateAmbassadorFederationRestored federationRestored	
RTI::FederateAmbassador.federationSaved	esimRTIfederateAmbassadorFederationSaved federationSaved	
RTI::FederateAmbassador.federationSynchronized	esimRTIfederateAmbassadorFederationSynchronized federationSynchronized	
RTI::FederateAmbassador.informAttributeOwnership	esimRTIfederateAmbassadorInformAttributeOwnership formAttributeOwnsership	
RTI::FederateAmbassador.initiateFederateRestore	esimRTIfederateAmbassadorInitiateFederateRestore initiateFederateRestore	
RTI::FederateAmbassador.initiateFederateSave	esimRTIfederateAmbassadorInitiateFederateSave initiateFederateSave	
RTI::FederateAmbassador.provideAttributeValueUpdate	esimRTIfederateAmbassadorProvideAttributeValueUpdate provideAttributeValueUpdate	
RTI::FederateAmbassador.receiveInteraction	esimRTIfederateAmbassadorReceiveInteraction receiveInteraction	
RTI::FederateAmbassador.reflectAttributeValues	esimRTIfederateAmbassadorReflectAttributeValues reflectAttributeValues	
RTI::FederateAmbassador.removeObjectInstance	esimRTIfederateAmbassadorRemoveObjectInstance removeObjectinstance	
RTI::FederateAmbassador.requestAttributeOwnershipAssumption	esimRTIfederateAmbassadorRequestAttributeOwnershipAssumption requestAttributeOwnershipAssumption	
RTI::FederateAmbassador.requestAttributeOwnershipRelease	esimRTIfederateAmbassadorRequestAttributeOwnershipRelease requestAttributeOwnershipRelease	
RTI::FederateAmbassador.requestFederationRestoreFailed	esimRTIfederateAmbassadorRequestFederationRestoreFailed initiateFederateRestoreFailed	
RTI::FederateAmbassador.requestFederationRestoreSucceeded	esimRTIfederateAmbassadorRequestFederationRestoreSucceeded initiateFederateRestoreSucceeded	

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	EsimRTI API	Remark
RTI::FederateAmbassador.requestRetraction	esimRTIfederateAmbassadorRequestRetraction requestRetraction	
RTI::FederateAmbassador.startRegistrationForObjectClass	esimRTIfederateAmbassadorStartRegistrationForObjectClass startRegistrationForObjectClass	
RTI::FederateAmbassador.stopRegistrationForObjectClass	esimRTIfederateAmbassadorStopRegistrationForObjectClass stopRegistrationForObjectClass	
RTI::FederateAmbassador.synchronizationPointRegistrationFailed	esimRTIfederateAmbassadorSynchronizationPointRegistrationFailed synchronizationPointRegistrationFailed	
RTI::FederateAmbassador.synchronizationPointRegistrationSucceeded	esimRTIfederateAmbassadorSynchronizationPointRegistrationSucceeded synchronizationPointRegistrationSucceeded	
RTI::FederateAmbassador.timeAdvanceGrant	esimRTIfederateAmbassadorTimeAdvanceGrant timeAdvanceGrant	
RTI::FederateAmbassador.timeConstrainedEnabled	esimRTIfederateAmbassadorTimeConstrainedEnabled timeConstrainedEnabled	
RTI::FederateAmbassador.timeRegulationEnabled	esimRTIfederateAmbassadorTimeRegulationEnabled timeRegulationEnabled	
RTI::FederateAmbassador.turnInteractionsOff	esimRTIfederateAmbassadorTurnInteractionsOff turnInteractionsOff	
RTI::FederateAmbassador.turnInteractionsOn	esimRTIfederateAmbassadorTurnInteractionsOn turnInteractionsOn	
RTI::FederateAmbassador.turnUpdatesOffForObjectInstance	esimRTIfederateAmbassadorTurnUpdatesOffForObjectInstance turnUpdatesOffForObjectInstance	
RTI::FederateAmbassador.turnUpdatesOnForObjectInstance	esimRTIfederateAmbassadorTurnUpdatesOnForObjectInstance turnUpdatesOnForObjectInstance	
RTI::FederateHandleSet<>	-	See RTI::<>HandleSet<>
RTI::ParameterHandleSet<>	-	See RTI::<>HandleSet<>
RTI::ParameterHandleValuePairSet<>	-	See RTI::<>HandleValuePairtSet<>
RTI::RTIambassador.attributeOwnershipAcquisition	esimRTIrtiAmbassadorAttributeOwnershipAcquisition attributeOwnershipAquisition	
RTI::RTIambassador.attributeOwnershipAcquisitionIfAvailable	esimRTIrtiAmbassadorAttributeOwnershipAcquisitionIfAvailable attributeOwnershipAquisitionIfAvailable	
RTI::RTIambassador.attributeOwnershipReleaseResponse	esimRTIrtiAmbassadorAttributeOwnershipReleaseResponse attributeOwnershipReleaseResponce	

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	EsimRTI API	Remark
RTI::RTIambassador.cancelAttributeOwnershipAcquisition	esimRTIrtiAmbassadorCancelAttributeOwnershipAcquisition cancelAttributeOwnershipAquisition	
RTI::RTIambassador.cancelNegotiatedAttributeOwnershipDivestiture	esimRTIrtiAmbassadorCancelNegotiatedAttributeOwnershipDivestiture cancelAttributeOwnershipDivestiture	
RTI::RTIambassador.changeAttributeOrderType	esimRTIrtiAmbassadorChangeAttributeOrderType changeAttributeOrderType	
RTI::RTIambassador.changeAttributeTransportationType	EsimRTIrtiAmbassadorChangeAttributeTransportationType changeAttributeTransportationType	
RTI::RTIambassador.changeInteractionOrderType	esimRTIrtiAmbassadorChangeInteractionOrderType changeInteractionOrderType	
RTI::RTIambassador.changeInteractionTransportationType	EsimRTIrtiAmbassadorChangeInteractionTransportationType changeInteractionTransportationType	
RTI::RTIambassador.createFederationExecution	esimRTIrtiAmbassadorCreateFederationExecution createFederationExecution	
RTI::RTIambassador.deleteObjectInstance	EsimRTIrtiAmbassadorDeleteObjectInstance deleteObjectInstance	
RTI::RTIambassador.destroyFederationExecution	esimRTIrtiAmbassadorDestroyFederationExecution destroyFederationExecution	
RTI::RTIambassador.disableAsynchronousDelivery	esimRTIrtiAmbassadorDisableAsynchronousDelivery disableAnsynchronousDelivery	
RTI::RTIambassador.disableAttributeRelevanceAdvisorySwitch	esimRTIrtiAmbassadorDisableAttributeRelevanceAdvisorySwitch disableAttributeRelevanceAdvisorySwitch	
RTI::RTIambassador.disableClassRelevanceAdvisorySwitch	esimRTIrtiAmbassadorDisableClassRelevanceAdvisorySwitch disableClassRelevanceAdvisorySwitch	
RTI::RTIambassador.disableInteractionRelevanceAdvisorySwitch	esimRTIrtiAmbassadorDisableInteractionRelevanceAdvisorySwitch disableInteractionRelevanceAdvisorySwitch	
RTI::RTIambassador.disableTimeConstrained	esimRTIrtiAmbassadorDisableTimeConstrained disableTimeConstrained	
RTI::RTIambassador.disableTimeRegulation	esimRTIrtiAmbassadorDisableTimeRegulation disableTimeRegulation	
RTI::RTIambassador.enableAsynchronousDelivery	esimRTIrtiAmbassadorEnableAsynchronousDelivery enableAnsynchronousDelivery	
RTI::RTIambassador.enableAttributeRelevanceAdvisorySwitch	esimRTIrtiAmbassadorEnableAttributeRelevanceAdvisorySwitch enableAttributeRelevanceAdvisorySwitch	

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	EsimRTI API	Remark
RTI::RTIambassador.enableClassRelevanceAdvisorySwitch	esimRTIrtiAmbassadorEnableClassRelevanceAdvisorySwitch enableClassRelevanceAdvisorySwitch	
RTI::RTIambassador.enableInteractionRelevanceAdvisorySwitch	esimRTIrtiAmbassadorEnableInteractionRelevanceAdvisorySwitch enableInteractionRelevanceAdvisorySwitch	
RTI::RTIambassador.enableTimeConstrained	esimRTIrtiAmbassadorEnableTimeConstrained enableTimeConstrained	
RTI::RTIambassador.enableTimeRegulation	esimRTIrtiAmbassadorEnableTimeRegulation enableTimeRegulation	
RTI::RTIambassador.federateRestoreComplete	esimRTIrtiAmbassadorFederateRestoreComplete federateRestoreComplete	
RTI::RTIambassador.federateRestoreNotComplete	esimRTIrtiAmbassadorFederateRestoreNotComplete federateRestoreNotComplete	
RTI::RTIambassador.federateSaveBegun	esimRTIrtiAmbassadorFederateSaveBegun federateSaveBegun	
RTI::RTIambassador.federateSaveComplete	esimRTIrtiAmbassadorFederateSaveComplete federateSaveComplete	
RTI::RTIambassador.federateSaveNotComplete	esimRTIrtiAmbassadorFederateSaveNotComplete federateSaveNotComplete	
RTI::RTIambassador.flushQueueRequest	esimRTIrtiAmbassadorFlushQueueRequest flushQueueRequest	
RTI::RTIambassador.getAttributeHandle	esimRTIrtiAmbassadorGetAttributeHandle getAttributeHandle	
RTI::RTIambassador.getAttributeName	esimRTIrtiAmbassadorGetAttributeName getAttributeName	
RTI::RTIambassador.getInteractionClassHandle	esimRTIrtiAmbassadorGetInteractionClassHandle getInteractionClassHandle	
RTI::RTIambassador.getInteractionClassName	esimRTIrtiAmbassadorGetInteractionClassName getInteractionClassName	
RTI::RTIambassador.getObjectClass	esimRTIrtiAmbassadorGetObjectClass getObjectClass	
RTI::RTIambassador.getObjectClassHandle	esimRTIrtiAmbassadorGetObjectClassHandle getObjectClassHandle	
RTI::RTIambassador.getObjectClassName	esimRTIrtiAmbassadorGetObjectClassName getObjectClassname	
RTI::RTIambassador.getObjectInstanceHandle	esimRTIrtiAmbassadorGetObjectInstanceHandle getObjectInstanceHandle	
RTI::RTIambassador.getObjectInstanceName	esimRTIrtiAmbassadorGetObjectInstanceName getObjectInstanceName	
RTI::RTIambassador.getOrderingHandle	esimRTIrtiAmbassadorGetOrderingHandle getOrderingHandle	
RTI::RTIambassador.getOrderingName	esimRTIrtiAmbassadorGetOrderingName getOrderingName	

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	EsimRTI API	Remark
RTI::RTIambassador.getParameterHandle	esimRTIrtiAmbassadorGetParameterHandle getParameterHandle	
RTI::RTIambassador.getParameterName	esimRTIrtiAmbassadorGetParameterName getParameterName	
RTI::RTIambassador.getTransportationHandle	esimRTIrtiAmbasadorGetTransportationHandle getTransportationHandle	
RTI::RTIambassador.getTransportationName	esimRTIrtiAmbassadorGetTransportationName getTransportationName	
RTI::RTIambassador.joinFederationExecution	esimRTIrtiAmbassadorJoinFederationExecution joinFederationExecution	
RTI::RTIambassador.localDeleteObjectInstance	esimRTIrtiAmbassadorLocalDeleteObjectInstance localDeleteObjectInstance	
RTI::RTIambassador.modifyLookahead	esimRTIrtiAmbassadorModifyLookahead modifyLookahead	
RTI::RTIambassador.negotiatedAttributeOwnershipDivestiture	esimRTIrtiAmbassadorNegotiatedAttributeOwnershipDivestiture negotiateAttributeOwnershipDivestiture	
RTI::RTIambassador.nextEventRequest	esimRTIrtiAmbassadorNextEventRequest nextEventRequest	
RTI::RTIambassador.nextEventRequestAvailable	esimRTIrtiAmbassadorNextEventRequestAvailable nextEventRequestAvailable	
RTI::RTIambassador.publishInteractionClass	esimRTIrtiAmbassadorPublishInteractionClass publishInteractionClass	
RTI::RTIambassador.publishObjectClass	esimRTIrtiAmbassadorPublishObjectClass publishObjectClass	
RTI::RTIambassador.queryAttributeOwnership	esimRTIrtiAmbassadorQueryAttributeOwnership queryAttributeOwnership	
RTI::RTIambassador.queryFederateTime	esimRTIrtiAmbassadorQueryFederateTime queryFederateTime	
RTI::RTIambassador.queryLBTS	esimRTIrtiAmbassadorQueryLBTSqueryLBTS	
RTI::RTIambassador.queryLookahead	esimRTIrtiAmbassadorQueryLookahead queryLookahead	
RTI::RTIambassador.queryMinNextEventTime	esimRTIrtiAmbassadorQueryMinNextEventTime queryMinNextEventTime	
RTI::RTIambassador.registerFederationSynchronizationPoint	esimRTIrtiAmbassadorRegisterFederationSynchronizationPoint registerFederationSynchronizationPoint	
RTI::RTIambassador.registerObjectInstance	esimRTIrtiAmbassadorRegisterObjectInstance registerobjectInstance	
RTI::RTIambassador.requestClassAttributeValueUpdate	esimRTIrtiAmbassadorRequestClassAttributeValueUpdate requestClassAttributeValueUpdate	
RTI::RTIambassador.requestFederationRestore	esimRTIrtiAmbassadorRequestFederationRestore requestFederationRestore	

Table I.14: RTI API mapped onto the EsimRTI API

RTI API	EsimRTI API	Remark
RTI::RTIambassador.requestFederationSave	esimRTIrtiAmbassadorRequestFederationSave requestFederationSave	
RTI::RTIambassador.requestObjectAttributeValueUpdate	esimRTIrtiAmbassadorRequestObjectAttributeValueUpdate requestObjectAttributeValueUpdate	
RTI::RTIambassador.resignFederationExecution	esimRTIrtiAmbassadorResignFederationExecution resignFederationExecution	
RTI::RTIambassador.retract	esimRTIrtiAmbassadorRetract retract	
RTI::RTIambassador.sendInteraction	EsimRTIrtiAmbassadorSendInteraction sendInteraction	
RTI::RTIambassador.subscribeInteractionClass	esimRTIrtiAmbassadorSubscribeInteractionClass subscribeInteractionClass	
RTI::RTIambassador.subscribeObjectClassAttributes	esimRTIrtiAmbassadorSubscribeObjectClassAttributes subscribeObjectClassAttributes	
RTI::RTIambassador.synchronizationPointAchieved	esimRTIrtiAmbassadorSynchronizationPointAchieved synchronizationPointAchieved	
RTI::RTIambassador.tick	esimRTIrtiAmbassadorTick tick	
RTI::RTIambassador.timeAdvanceRequest	esimRTIrtiAmbassadorTimeAdvanceRequest timeAdvanceRequest	
RTI::RTIambassador.timeAdvanceRequestAvailable	esimRTIrtiAmbassadorTimeAdvanceRequestAvailable timeAdvanceRequestAvailable	
RTI::RTIambassador.unconditionalAttributeOwnershipDivestiture	esimRTIrtiAmbassadorUnconditionalAttributeOwnershipDivestiture unconditionalAttributeOwnershipDivestiture	
- RTI::RTIambassador.unpublishInteractionClass	esimRTIrtiAmbassadorUnpublishInteractionClass unpublishInteractionClass	
RTI::RTIambassador.unpublishObjectClass	esimRTIrtiAmbassadorUnpublishObjectClass unpublishObjectClass	
RTI::RTIambassador.unsubscribeInteractionClass	esimRTIrtiAmbassadorUnsubscribeInteractionClass unsubscribeInteractionClass	
RTI::RTIambassador.unsubscribeObjectClass	esimRTIrtiAmbassadorUnsubscribeObjectClass unsubscribeObjectClass	
RTI::RTIambassador.updateAttributeValues	esimRTIrtiAmbassadorUpdateAttributeValues updateAttributeValues	

Table I.14: RTI API mapped onto the EsimRTI API

Appendix J

Run-time Interface Description

J.1 Introduction

The run-time interface of EuroSim is the interface which is used to communicate with the running simulator. The Simulation Controller tool and the batch utility use this interface to start a new simulation run and to control it.
This interface description provides a step by step description of how to start the simulator and what commands to send to control the simulator once it is running. The order of the chapters is the order of each step.
In Section J.2 is explained how to start a simulator using the EuroSim daemon and how to connect to the new simulator. In order to receive autonomous messages from the simulator the client must subscribe to certain channels. This is explained in Section J.3. The following 4 chapters describe each one channel. Shutdown and cleanup is described in Section J.8. Finally, Section J.9, gives an overview of the available manual pages on the subject.

J.2 Simulator start-up

On each host where a EuroSim simulation can run, a daemon must be started. This daemon is responsible for the starting of simulators (among other things). The interface to this RPC daemon is defined in `esimd.x` in `$EFOROOT/include/rpcsvc`. The header file can be found in `$EFOROOT/include/esim`. The details of the interface are described in manual page *esimd(3)*.
With the advent of EuroSim Mk3 a new version of the interface was created to support the new simulation definition file. The RPC daemon still supports the old version. In this document only the new interface is described.
To start a simulator the RPC call `start_session_3()` must be done. The EuroSim daemon running on a EuroSim simulator host will launch the actual simulator executable. This call takes the following structure as argument:

Listing J.1: session3_def structure

```
struct session3_def {
      file_def sim;
      char *work_dir;
      char *simulator;
      file_def schedule;
      struct {
            u_int scenarios_len;
            file_def *scenarios_val;
      } scenarios;
      char *dict;
      file_def model;
      char *recorderdir;
      struct {
            u_int initconds_len;
            file_def *initconds_val;
      } initconds;
      char *exports;
      struct {
            u_int environment_len;
            env_item *environment_val;
      } environment;
      int prefcon;
      int umask;
      int flags;
};
struct file_def {
      char *path;
      char *vers;
};
```

The `file_def` structure is used to store the name of the file and an optional version string. Table J.1 describes each member of the `session3_def` structure.

field	**description**
`sim`	The path and version name of the simulation definition file (.sim). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.
`work_dir`	The path name of the current working directory of the simulator. The directory should exist and be accessible by the EuroSim daemon. Normally this is done by making the directory available through NFS in case the RPC call is performed from a different host.
`simulator`	The file name of the simulator executable (.exe). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.
`schedule`	The file name of the simulator schedule file (.sched). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.
`scenarios`	An array of scenario files (.mdl). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.
`dict`	The file name of the data dictionary file (.dict). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.
`model`	The file name of the model file (.model). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`. This file is not actually used by the simulator for reading. It used for tracing purposes as a reference.
`recorderdir`	the path name of the directory where all recordings are stored. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.
`initconds`	An array of initial condition files (.init). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.
`exports`	the file name of the exports file (.export). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in `work_dir`.

Table J.1: session_def structure

field	description
`environment`	an array of environment variables in the usual format `VAR=value`. Normally it is sufficient to copy the entire current environment into this array. If you want to start the simulator with a custom environment setting you have to set at least the following environment variables used by EuroSim in addition to the ones used by the simulator model software. `EFOROOT` should be set to the EuroSim installation directory. `EFO_SHAREDMEMSIZE` is the amount of memory reserved for dynamic memory allocation. Default is 4194304 (4 MB). This value can be set in the ModelEditor since Mk3rev2. `EFO_STACKSIZE` is the stack size reserved for each thread of the simulator. Default is 128k (IRIX) or 16k (Linux). This value can be set in the ModelEditor since Mk3rev2. `PWD` is the current working directory and is set to `work_dir` by the daemon if it is not present. `LD_LIBRARYN32_PATH` should be set to the path of the shared libraries of EuroSim for IRIX 6.5. The value is normally `$EFOROOT/lib32`. `LD_LIBRARY_PATH` should be set to the path of the shared libraries of EuroSim for other systems than IRIX 6.5 (e.g. Linux). The value is normally `$EFOROOT/lib`.
`prefcon`	set to -1 under normal circumstances, a connection number is selected by the daemon and returned on successful start-up of the simulator. Put a positive value here if you want to force the new simulator to have a specific connection number.
`umask`	the umask used for creation of new files. See *umask(2)*.
`flags`	there are currently two flags defined: SESSION_REALTIME and SESSION_NO_AUTO_INIT. Flags shall be or-ed together. Add the SESSION_REALTIME flag for real-time runs, or do not set the flag for non-real-time runs. The SESSION_NO_AUTO_INIT flag can be set to prevent the EuroSim scheduler from automatically going into initializing state. This is used by the EuroSim Simulation Controller to set break points and traces and to disable tasks before the simulation goes into initializing state.

Table J.1: session_def structure

The following small example in C will show how to start a simulator using representative values for the parameters.

Listing J.2: tc_example.c

```
#include <rpc/rpc.h>
#define _RPCGEN_CLNT
#include <esimd.h>

int main(void)
{
  struct session3_def session;
  struct start_result *result;
  env_item env[6];
  file_def scenario;
  file_def initcond;
  CLIENT *clnt;

  session.sim.path="Demo.sim";
  session.sim.vers="";
  session.work_dir="/home/user/projects/STD";
  session.simulator="Demo.exe";
  session.schedule.path="Demo.sched";
  session.schedule.vers="";
  scenario.path="Demo.mdl";
  scenario.vers="";
  session.scenarios.scenarios_len=1;
  session.scenarios.scenarios_val=&scenario;
  session.dict="Demo.dict";
  session.model.path="Demo.model";
  session.model.vers="";
  session.recorderdir="2000-04-01/00:00:01";
  initcond.path="Demo.init";
  initcond.vers="";
  session.initconds.initconds_len=1;
  session.initconds.initconds_val=&initcond;
  session.exports="Demo.exports";
  session.prefcon=-1;
  session.umask=022;
  session.flags=SESSION_REALTIME;
  env[0] = "LD_LIBRARY_PATH=/usr/EuroSim/lib";
  env[1] = "HOME=/home/user";
  env[2] = "EFO_HOME=/home/user/project/EfoHome";
  env[3] = "LD_LIBRARYN32_PATH=/usr/EuroSim/lib32";
  env[4] = "PWD=/home/user/project/STD";
  env[5] = "EFOROOT=/usr/EuroSim";
  session.environment.environment_len = 6;
  session.environment.environment_val = env;

  clnt = clnt_create("spiff", ESIM_PROG, ESIM_VERS3, "tcp");
  clnt->cl_auth = authunix_create_default();
  result = start_session_3(&session, clnt);
  if (result->status == ST_SUCCESS) {
   printf("simulator started at connection %d\n",
         result->start_result_u.prefcon);
   return 0;
  }
  else {
   error_array *errors;
   unsigned int i;

   printf("simulator failed to start:\n");
   errors = &result->start_result_u.errors;
   for (i = 0; i < errors->error_array_len; i++) {
    printf("%s\n", errors->error_array_val[i]);
   }
   xdr_free((xdrproc_t)xdr_start_result, (char*)result);
   return 1;
  }
}
```

The above example can be compiled as follows:
IRIX:

```
cc -I$EFOROOT/include/esim -o tc_example tc_example.c \
   -L$EFOROOT/lib32 -les
```

Linux:

```
cc -I$EFOROOT/include/esim -o tc_example tc_example.c \
   -L$EFOROOT/lib -les -lpthreads -lposixtime
```

It is however easier if you use the `esimd_complete_session` function to fill in the missing pieces. You only have to provide the simulation definition and the other entries will be completed by the `esimd_complete_session`:

Listing J.3: Example of the use of esimd_complete_session()

```
int main(void)
{
  struct session3_def session;
  struct start_result *result;
  CLIENT *clnt;
  extern char **environ;

  memset(&session, 0, sizeof(session));
  session.work_dir = ds("/home/user/projects/STD");
  session.sim.path = ds("Demo.sim");
  if (esimd_complete_session(&session, environ) == 0) {
    /* start session */
    esimd_free_session(&session);
  }
}
```

After successfully launching the simulator a connection can be created. There is some time between launching the simulator and when a connection can be created. This time is normally less than a second. To make a connection with the simulator the function `eventConnect()` must be called.

Listing J.4: Connect to a simulator

```
Connection *conn;

conn = eventConnect("spiff", "test-controller", connClient,
                    eventHandler, userdata, true,
                    result->start_result_u.prefcon);
```

The second parameter is the client name. In this example it has the value "test-controller". If the string contains the sub-string "-observer", the client is treated as a read-only client of the simulator. The client can monitor variables but not change them. The client cannot do anything which influences the simulator. The parameter `eventHandler` is the callback function which is called when an event from the simulator has been received. Each event results in one call to the callback. The callback must determine the type of the message and decode its contents. The callback has one parameter called `userdata` which contains the value given when calling `eventConnect()`.
The following example in C code shows an implementation of the `eventHandler` callback.

Listing J.5: Example of an eventHandler callback function

```
int eventHandler(Connection *conn, const evEvent *event,
                 void *userdata)
{
  size_t offset = evEventArgOffset();
  int sev;
  char *mesg;
  double speed;
  AuxTime simtime, wallclocktime;

  simtime = evEventSimTime(event);
  wallclocktime = evEventRunTime(event);

  switch (evEventType(event)) {
  case maMessage:
    evEventArg(event, &offset, EV_ARG_STRING(&sev));
    evEventArg(event, &offset, EV_ARG_STRING(&mesg));
    printf("%s: %s\n", auxSeverity2string(sev), mesg);
    break;

  case scSpeed:
    evEventArg(event, &offset, EV_ARG_DOUBLE(&speed));
    printf("speed = %f\n", speed);
    break;
  }
  return 0;
}
```

The programmer can choose to use synchronous or asynchronous handling of events. The example above has chosen for asynchronous event handling. This means that each time an event arrives a signal (SIGIO) is sent to the application. The library installs a signal handler which ultimately calls the `eventHandler` callback. If you select synchronous handling, the application has full control over when events are read. Using *select(2)* the programmer can determine if data is ready to be read and would then call `eventPoll()` to process all the available events. The function `eventPoll()` will call the `eventHandler` callback for each event.

J.3 Subscribing to channels

After connecting to the server, the simulator client can subscribe to several channels. When a client is subscribed to a channel it will receive events that are sent automatically without a previous client request. These messages are either generated by the models or by the simulator infrastructure. Each channel addresses a specific area of interest. At the moment of subscribing (or joining) a channel, the client will receive a number of messages describing the current state relating to that channel. The messages after joining a channel are described in the chapter dedicated to that channel. Table J.2 describes each channel.

Channel	Channel Identifier	Define	Chapter
Real time control	rt-control	CONTROLCHANNEL	Section J.4
Mission	mdlAndActions	MISSIONCHANNEL	Section J.5
Monitor	data-monitor	MONITORCHANNEL	Section J.6
Scheduler control	sched-control	SCHEDCONTROLCHANNEL	Section J.7

Table J.2: Channel Descriptions

To subscribe to a channel the function `eventJoinChannel()` must be called:

Listing J.6: Join a channel

```
Connection *conn; /* must be set with eventConnect() */

eventJoinChannel(conn, MISSIONCHANNEL);
```

For a simulator client it is mandatory to join the MISSIONCHANNEL. After launching the simulator, the simulator waits with the further initialization until the first client joins this particular channel. The simulator can then send its messages to a client. This is particular useful when something goes wrong. It enables the user to read the messages and take corrective actions.
The following four chapters will describe each channel in detail. Each chapter will contain tables describing events coming from the simulator and commands which can be sent to the simulator. Each command is sent using an `event*` macro. This macro takes 2 or more arguments. The first two arguments are always the same. The first argument is the handle to the connection, the second argument is a pointer to an `AuxStamp` structure which can be a NULL pointer for clients.

J.4 Real time control channel

The real time control channel is used to request and report state changes of the simulator. The simulator client can request state changes and the simulator will report the new state as soon as it has been reached.

Figure J.1 shows the state transition diagram applicable to an external application controlling the simulator. Next to the arrows are the functions to be called for each state transition. In the boxes the name of the event is shown that is sent to the client when entering the state. The only exception is the `eventReset` command. This command performs a small scenario consisting of a state transition from standby state to exiting state. from exiting to unconfigured, from unconfigured to initializing. In initializing state the automatic state transition to standby is performed as specified in the schedule.

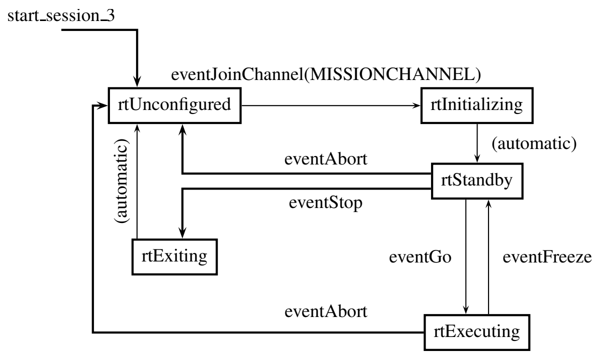


Figure J.1: Simulator states

Table J.3 lists the messages sent to the client after joining the real-time control channel.

Event	**Description**	**Arguments**
rtUnconfigured, rtInitializing, rtStandby, rtExecuting, rtExiting	Current state	-
rtMainCycle	Main cycle time of schedule	cycle time in timespec format (tv_sec and tv_nsec)

Table J.3: Real time control channel join events

Table J.4 shows the functions which can be used to request the change and the events sent back as a result.

Command	**Description**	**Response**
eventFreeze	Request state transition to standby state from initializing or executing state	rtStandby
eventFreezeAt <*wallclocktime*>	Same as previous but wait until a certain wallclock time.	rtStandby
eventFreezeAtSimtime <*simtime*>	Same as previous but wait until a certain simulation time	rtStandby
eventGo	Request state transition to executing state from standby state	rtExecuting

Table J.4: Real time control channel commands

Command	**Description**	**Response**
eventGoAt <*wallclocktime*>	Same as previous but wait until a certain wallclock time.	rtExecuting
eventStep	Request the execution of one main cycle.	rtExecuting rtStandby
eventReset	Request reinitialization from standby state	rtExiting rtUnconfigured rtInitialising rtStandby (if performed automatically by the schedule configuration)
eventStop	Request the controlled termination from standby state.	rtExiting rtUnconfigured
eventAbort	Request immediate abort from any state.	rtUnconfigured
eventHealth	Request health check.	maMessage <*eurosimversion*> maMessage <*executable* `is healthy.`> maMessage <`executing "`*scenario*`" for "`*group*`"`> rtHealth

Table J.4: Real time control channel commands

As state transitions may take some time, a `rtTimeToNextState` message is sent to the simulator client which contains the amount of time to the transition.
After joining the rt-control channel the current simulator state is sent. All state transitions from then on are sent to the client, including automatic state transitions, or transitions requested by another client. The standard time stamps of the state transition message can be used to calculate valid future state transition times which can be used to issue timed state transition commands. To calculate a valid future transition time take the wallclock or simulation time from the last state transition message and add an integer number of main cycle times.
The following example in C requests a state transition at midnight on April 1, 2001 (wallclock time):

Listing J.7: Time state transition

```
Connection *conn; /* must be set with eventConnect() */
struct timespec tv;
struct tm tm;

tm.tm_sec = 0;
tm.tm_min = 0;
tm.tm_hour = 0;
tm.tm_mday = 1;
tm.tm_mon = 4;
tm.tm_year = 100; /* years since 1900 */
tm.tm_isdst = 0;
tv.tv_sec = mktime(&tm);
tv.tv_nsec = 0;
eventGoAt(conn, NULL, &tv);
```

At the indicated time an event `rtExecuting` is sent to the simulator client.

J.5 Mission channel

The mission channel is used for all activities relating to the manipulation of scenarios and actions. Scenarios are either loaded at start-up from disk or are created on the fly using the commands listed in this chapter. Scenarios loaded from disk can be modified in the simulator. The changes are only in the running simulator, not in the file on disk.
Table J.5 lists the messages sent to the client after joining the mission channel.

Event	**Description**	**Arguments**
maDenyWriteAccess	Write access notification.	`on/off`
maCurrentWorkingDir	Working directory notification.	working directory
maCurrentDict	Current data dictionary notification.	dictionary file name
maSimDef	Current simulation definition file	simulation definition filename
maCurrentResultDir	Current result directory notification.	result directory
maCurrentCycletime	Current action manager cycle time notification.	cycle time
maCurrentTimeMode	Current time mode notification.	0 = relative 1 = UTC
maCurrentInitconds	Current list of initial condition files notification.	simulation definition file, initial condition file(s)
maRecording	Recording status notification.	`on/off`

Table J.5: Mission channel join events

Table J.6 shows the events which can be sent to the simulator and the responses they send back. Arguments are enclosed in angled brackets. Literal messages are in courier where variant parts are in italic. Wherever you see the word *file* (as in scenario file) a file on disk is meant. All other references to scenario are to the run-time data structure inside the simulator.

Command	**Description**	**Response**
eventNewMission <*scenario*>	Create a new (virtual) scenario	maNewMission <*scenario*> maMessage <`scenario "`*scenario*`" created for "`*group*`"`>
eventOpenMission <*scenariofile*>	Open an existing scenario file	maOpenMission <*scenariofile*> maMessage <`scenario "`*scenariofile*`" opened for "`*group*`"`>
eventCloseMission <*scenario*>	Close a scenario	maMessage <`scenario "`*scenario*`" owned by "`*group*`" closed`> maCloseMission <*scenario*>
eventNewAction <*scenario*> <*actiontext*>	Create a new action in a scenario	maNewAction <*scenario*> <*actionname*> maMessage <`new active action "`*actionname*`" in "`*scenario*`"`>
eventDeleteAction <*scenario*> <*actionname*>	Delete an action in a scenario	maDeleteAction <*scenario*> <*action*> maMessage <`deleted action "`*action*`" from "`*scenario*`"`>

Table J.6: Mission channel commands

Command	**Description**	**Response**
eventActionExecute <*scenario*> <*actionname*>	Execute (trigger) an action in a scenario	maActionExecute <*scenario*> <*action*> maActionExecuteStop <*scenario*> <*action*> maMessage <`manually triggered action "`*action*`"`>[1]
eventActionActivate <*scenario*> <*actionname*>	Make an action active in a scenario	maActionActivate <*scenario*> <*action*> maMessage <`action "`*action*`" activated`>
eventActionDeActivate <*scenario*> <*actionname*>	Make an action inactive in a scenario	maActionDeActivate <*scenario*> <*action*> maMessage <`action "`*action*`" deactivated`>
eventCurrentInitconds <*simulation definition*> <*initconds list*>	Sets a new list of initial conditions files.	maCurrentInitconds <*simulation definition*> <*initial condition file(s)*>
eventSnapshot <*filename*> <*comment*>	Make a snapshot.	maMessage <`snapshot made for` *filename*> maSnapshot <*snapshot filename*> <*comment*>
eventReload <*snapshot filename*> <*set simtime*>	Reload a snapshot file. The second argument *set simtime* can be set to on or off. When it set to on, the simulation time is set to the value present in the snapshot file.	maReload <*snapshot filename*> <*set simtime*> If snapshot is loaded with simtime: scSimtime <*simtime*> <*wallclocktime*> maMessage <`new simulation time:` *simtime*> In all cases: maMessage <`loaded` *filename*`:` *comment*>
eventMark <*marktext*> <*number*>	Create a mark.	maMark <*mark string*> <*mark count*>
eventMessage <*text*>	Send a message to the simulator client	maMessage <*text*>
eventRecording <*on/off*>	Suspend/resume recording.	When switching off: maRecording <*off*> maMessage <`suspended recordings`> When switching on: maRecording <*on*> maMessage <`resumed recordings`>
eventRecordingSwitch	Switch recorder files.	For each recorder file: maRecorderFileClosed <*recorderfilename*> maMessage <`Switching recorder files`>

Table J.6: Mission channel commands

Table J.7 shows the events relating to messages which can be sent from the model code or the simulator infrastructure.

[1] In case a monitor action (obsolescent) is executed various messages from the monitor channel are generated. These can be found in Table J.9

Event	Description	Arguments
maMessage	Message	severity, message

Table J.7: Message events

Table J.8 shows the messages sent autonomously every 2 seconds.

Event	Description	Arguments
maRecordingBandwidth	Current recording bandwidth consumption notification.	bandwidth (bytes/sec)
maStimulatorBandwidth	Current stimulator bandwidth consumption notification.	bandwidth (bytes/sec)

Table J.8: Mission channel autonomous messages

The following example in C requests the loading of a scenario file into the simulator.

Listing J.8: Load an MDL file into the simulator

```
Connection *conn; /* must be set with eventConnect() */

eventOpenMission(conn, NULL, "/home/eurosim/project/proj.mdl");
```

The result will be a `maMessage` event informing about the successful opening of the scenario file.

J.6 Monitor channel

The monitor channel is used to manipulate monitors. Table J.9 shows the messages which are sent when triggering a monitor action (obsolescent). The event `dtMonitor` is sent at the start to mark the beginning of a new monitor. If one monitor action monitors multiple variables, the `dtMonitorVar` event is sent once for each variable. The event `dtMonitorDone` ends the list. The client application can then set up the display for the new monitor. The client must send a `eventAdd2LogList` command for each variable. After that every 0.5 seconds (2 Hz) an update (`dtLogValueUpdate`) is sent from the simulator to the client.
The frequency can be changed to a higher or a lower frequency by passing an option -f to the EuroSim daemon esimd with the required frequency. Using this command line option for the daemon sets the frequency for all simulators. The frequency may be a floating point number.
The frequency can be changed to a lower frequency by passing an option -d to the EuroSim daemon with a divisor. This option reduces the frequency of the monitor updates etc. with the specified integer factor. This option affects all simulators started with the daemon.
The order of messages periodically sent by the simulator to the client is as follows:

- monitor values
- heartbeat
- cpu load (optionally)

Alternatively it is possible to retrieve the value of a variable only once by using the `dtGetValueRequest` command.

Event	Description	Arguments
dtMonitor	Start new monitor	scenario, action name

Table J.9: Monitor events on monitor action (obsolescent) execution

Event	Description	Arguments
dtMonitorVar	Monitor variable	variable name
dtMonitorDone	Finish new monitor	attributes

Table J.9: Monitor events on monitor action (obsolescent) execution

Table J.10 shows the event sent periodically at 2 Hz for each variable in an active monitor.

Event	Description	Arguments
dtLogValueUpdate	Monitor value update	variable name, value

Table J.10: Monitor update event

Table J.11 shows the commands which can be sent.

Command	Description	Response
eventAdd2LogList <*variable name*>	Add variable to list of monitored variables	dtAdd2LogList <*variable name*> dtLogValueUpdate <*variable*> <*value*>
eventRemoveFromLogList <*variable name*>	Remove variable from list of monitored variables	dtRemoveFromLogList <*variable name*>
eventSetValueRequest <*variable name*> <*value*>	Set variable to value	dtSetValueRequest <*variable name*> <*value*> maMessage <`set "`*variable*`" to "`*value*`"`> If variable is monitored at that moment: dtLogValueUpdate <*variable*> <*value*>
eventCpuLoadSetPeak <*processor*> <*peak_time (ms)*>	Monitor the CPU load of a specific CPU	dtCpuLoadSetPeak <*processor*> <*peak_time (ms)*> At a frequency of 2 Hz: dtCpuLoad <*processor*> <*average*> <*peak*>
dtGetValueRequest <*variable name*>	Get the value of a variable once	dtLogValueUpdate <*variable name*> <*value*>

Table J.11: Monitor channel commands

Table J.12 shows the messages sent on the mission channel autonomously with a frequency of 2 Hz.

Event	Description	Arguments
dtHeartBeat	Heartbeat	count

Table J.12: Monitor channel autonomous events

The following example in C requests the monitoring of a specific variable in the data dictionary of the running simulator.

Listing J.9: Start monitoring a variable

```
Connection *conn; /* must be set with eventConnect() */
```

```
eventAdd2LogList(conn, NULL, "/model/file/var");
```

From this moment on `dtLogValueUpdate` messages will be sent to the simulator client with 2 Hz. To stop these messages call:

Listing J.10: Stop monitoring a variable

```
eventRemoveFromLogList(conn, NULL, "/model/file/var");
```

J.7 Scheduler control channel

The scheduler control channel is used to manipulate and monitor the EuroSim scheduler. Table J.13 lists the messages sent to the client after joining the mission channel.

Event	**Description**	**Arguments**
scTaskListStart	Beginning of task list	-
scTaskStart	Beginning of entry point list of a task	*taskname*, *enabled*
scTaskEntry	Entry point description	*entryname*, *breakpoint*, *trace*
scTaskEnd	End of entry point list of a task	-
scTaskListEnd	End of task list	-
scEventListStart	Beginning of event list	-
scEventInfo	Event description	*eventname*, *state*, *is_standard*
scEventListEnd	End of event list	-
scGoRT	Real-time mode notification	*enable*

Table J.13: Scheduler control join events

Table J.14 lists the available commands.

Command	**Description**	**Response**
eventSetBrk <*taskname*> <*entrynr*> <*enable*>	Set breakpoint The where-list is only sent if the simulator state is rtExecuting.	scSetBrk <*taskname*> <*entrynr*> <*enable*> maMessage <`debugging task: break on task "`*task*`" entry "`*entrypoint*`" enabled/disabled`> scWhereListStart scWhereEntry <*taskname*> <*entrynr*> scWhereListEnd
eventStepTsk	Step to next entry point	scWhereListStart scWhereListEnd scStepTsk scWhereListStart scWhereEntry <*taskname*> <*entrynr*> scWhereListEnd maMessage <`STEP on` *task*`:`*entrypoint*>

Table J.14: Scheduler control commands

Command	**Description**	**Response**
eventContinue	Continue execution up to next breakpoint	scWhereListStart scWhereListEnd scContinue scWhereListStart scWhereEntry <*taskname*> <*entrynr*> scWhereListEnd
eventGoRT <*enable*>	Switch between real-time and non-real-time.	scGoRT <*enable*>
eventListTasks	Request task list	scTaskListStartscTaskStart <*taskname*> <*enabled*> scTaskEntry <*entryname*> <*breakpoint*> <*trace*> scTaskEndscTaskListEnd
eventTaskDisable <*taskname*> <*disable*>	Disable a task	scTaskDisable <*taskname*> <*disable*> maMessage <`task "`*taskname*`" disabled`/`enabled`>
eventSetTrc <*taskname*> <*entrynr*> <*enable*>	Enable/disable tracing of an entry point	scSetTrc <*task.entrypoint*> <*enable*>
eventClearBrks	Clear all breakpoints	scClearBrks
eventClearTrcs	Clear all traces	scClearTrcs
eventWhere	Request current position in schedule	scWhereListStart scWhereEntry <*taskname*> <*entrynr*> scWhereListEnd
eventListEvents	Request event list	scEventListStartscEventInfo <*eventname*> <*state*> <*is_standard*> scEventListEnd
eventRaiseEvent <*event*>	Raise event	scRaiseEvent <*event*>
eventSimtime <*simtime*>	Set simulation time	scSimtime <*simtime*> maMessage <`new simulation time:` *simtime*>
eventRaiseEventAt <*event*> <*sec*> <*nsec*>	Raise event at wallclock time	scRaiseEventAt <*event*> <*sec*> <*nsec*>
eventRaiseEventAtSimtime <*event*> <*sec*> <*nsec*>	Raise event at simulation time	scRaiseEventAtSimtime <*event*> <*sec*> <*nsec*>
eventSpeed <*speed*>	Set relative clock speed Only when running non-realtime. When speed is set to -1, the simulator will run as fast as possible.	scSpeed <*speed*>

Table J.14: Scheduler control commands

There are three groups of events which need additional attention. These groups of events are used to transmit complicated data structures to the client:

- task list
- event list

- debugger position list

The task list uses 5 events which are sent in a nested fashion.
The task list starts with `scTaskListStart` and ends with `scTaskListEnd`. After `scTaskListStart` one or more tasks are sent. Each task starts with `scTaskStart` and ends with `scTaskEnd`. After `scTaskStart` one or more entry points are sent. Each entry point is sent using `scTaskEntry`.
The event list starts with `scEventListStart` and ends with `scEventListEnd`.
After `scEventListStart` one or more event descriptions are sent. Each event is sent using `scEventInfo`.
The debugger position list, also called *where* list, starts with `scWhereListStart` and ends with the response `scWhereListEnd`. After `scWhereListStart` zero or more positions are sent. Each position is sent using `scWhereEntry`.
Table J.15 shows the messages sent autonomously every 2 seconds.

Event	**Description**	**Arguments**
scSpeed	Relative clock speed. Only when running non-real-time.	speed

Table J.15: Scheduler control autonomous messages

The following example in C sets the speed of a non-realtime simulator to run as fast as possible.

Listing J.11: Let the simulator run as fast as possible

```
Connection *conn; /* must be set with eventConnect() */

eventSpeed(conn, NULL, -1);
```

From now the scheduler from EuroSim will execute all models as fast as possible. The event `scSpeed` is sent at 2 Hz. The value of the speed parameter will reflect the actual acceleration achieved.

J.8 Simulator shutdown

At the moment a simulator executable exits, all clients are automatically disconnected. In that case event `evShutdown` is received. This is a pseudo event which is not sent by the simulator but is generated as soon as a socket shutdown is detected. The socket has been destroyed by then and it is not possible to send messages to the simulator anymore.
It is also possible to actively terminate the simulator connection by calling `eventDisconnect()`:

Listing J.12: Disconnect from the simulator

```
Connection *conn; /* must be set with eventConnect() */

eventDisconnect(conn);
```

After disconnecting from the simulator it is not possible to send messages to the simulator. However it is possible to reconnect to the simulator using the functions described in Section J.2.

J.9 Manual pages

Table J.16 shows an overview of the on-line available manual pages of EuroSim. These pages are the ultimate reference for all events.

Man Page	**Description**
events(3)	Retrieval system for information about all available EuroSim events

Table J.16: Overview of relevant manual pages.

Man Page	**Description**
evEvent(3)	Event construction, access and I/O functions
rt-control(3)	Real-time control events
data-monitor(3)	Monitor events
sched-control(3)	Scheduler control events
mdlAndActions(3)	Scenario events
esimd(3)	EuroSim daemon RPC client interface functions and types
evc(3)	Functions for clients to setup multi bi-directional event driven connections
evHandler(3)	Functions for server and client to create handlers for incoming events
extClient(3)	Functions for an external client to establish and control access to a EuroSim simulator
extView(3)	Functions to create, control and destroy data views.
extMdl(3)	Functions for an external client to manage scenarios and actions running on a EuroSim simulator
extMessage(3)	Functions for an external client to send messages to a EuroSim simulator.
esimLink(3)	Functions for creating and manipulating simulated satellite communication links

Table J.16: Overview of relevant manual pages.

Appendix K

Scheduler behavior with as fast as possible simulation

K.1 Introduction

The execution sequence of as fast as possible (AFAP) scheduling is a result of the same constraints as normal real-time scheduling and the overall behavior will thus be the same. However, one should be aware that AFAP scheduling exploits the parallelism of the schedule to a maximum. If a schedule is not well defined, this parallelism could lead to erroneous behavior. Below is an explanation of the operation of the scheduler, followed by some examples illustrating AFAP scheduling and some consequences regarding parallelism.

K.2 Deadlines and simulation time

A task in EuroSim has a deadline which is equal to the sum of its start time and its allowed execution time. A deadline is the point in time at which a task should be ready. In a non real-time simulation the deadline is not a real world time, but a (virtual) simulation time. In a normal speed non real-time simulation this simulation time runs as fast as the real world time. However when a task is not ready before its deadline, the simulation time is halted until the task gets ready. Thus, when a task misses a deadline no more tasks will be started until that task gets ready.
When the scheduler is running a simulation as fast as possible it increments the simulation time and starts tasks, until the simulation time reaches the deadline of one of the started tasks. The scheduler then waits until that task is ready and continues to increment the simulation time until the next deadline is reached.

K.3 Example 1: AFAP simulation with 2 independent tasks

Two tasks A and B are scheduled according to the schedule of Figure K.1. Both tasks have an allowed execution time of 15 ms. Task A has a real execution time of 4 ms and runs on processor 1. Task B has a real execution time of 6 ms and runs on processor 2. The real time execution sequence is shown in Figure K.2. Tasks B starts after task A is ready.

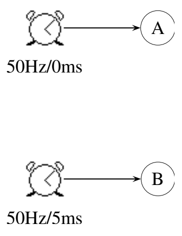


Figure K.1: Schedule of example 1

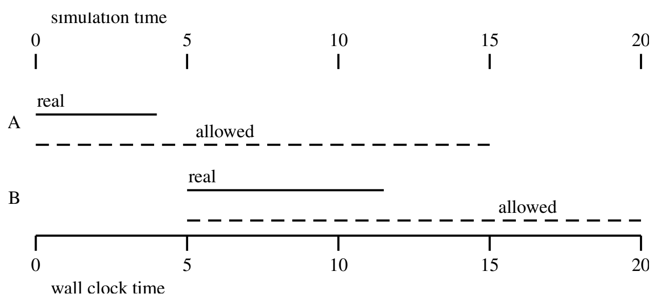


Figure K.2: Real time execution sequence

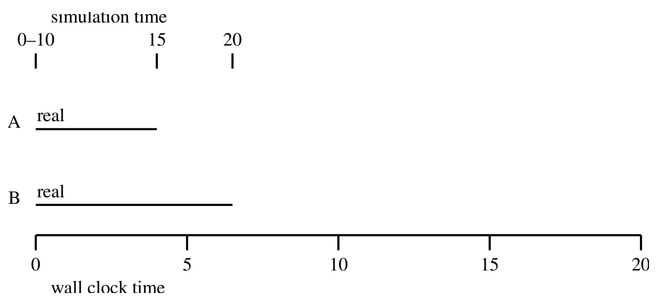


Figure K.3: AFAP execution sequence

Figure K.3 shows the execution sequence of the AFAP simulation. After task A is started, the simulation time may be increased immediately up to 5 ms, because there is no task with a deadline at 5 ms. Task B can thus be started and the simulation time can be increased up to 10 ms. The simulation time can be increased up to 15 ms only after the completion of task A and up to 20 ms after the completion of task B. The 20 ms of simulation time are executed in 6 ms real time, an acceleration factor of 3.3.
In the AFAP simulation task A and B run in parallel where they were running exclusive in the real time simulation.

K.4 Example 2: implicit mutual exclusion of two tasks

Tasks A and B are scheduled as in example 1. However, the allowed execution time for task A is set to 5 ms. The real time execution shown in Figure K.4 does not differ from Figure K.2. But, the parallelism in the AFAP simulation (Figure K.5) has disappeared. The simulation time cannot be incremented up to 5 ms until task A has completed.
Due to this implicit exclusion the acceleration factor is 2.

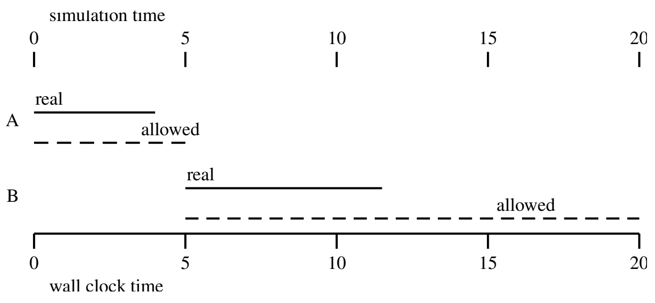


Figure K.4: Real time execution sequence with 5 ms allowed execution time for task A

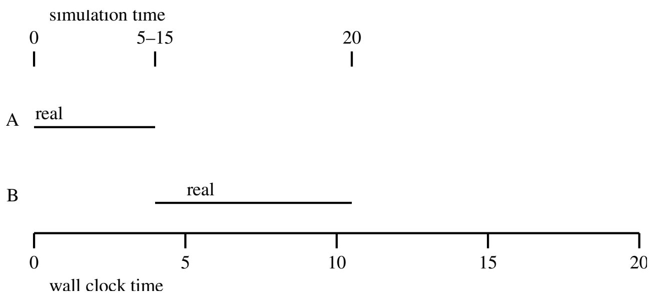


Figure K.5: AFAP execution sequence with 5 ms allowed execution time for task A

K.5 Example 3: A chain of tasks is a pipeline and has parallelism

A chain of tasks as shown in Figure K.6 is a pipeline and will be executed as such by the scheduler.

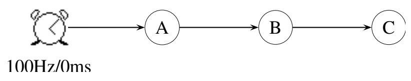


Figure K.6: A chain of tasks forming a pipeline

The schedule has a basic frequency of 1000 Hz and the tasks have the following properties:

- Processor: any (Schedule Editor default)
- Allowed execution time: 4 ms
- Real execution time: 3 ms

In a real time run these specifications result in the following task sequence:

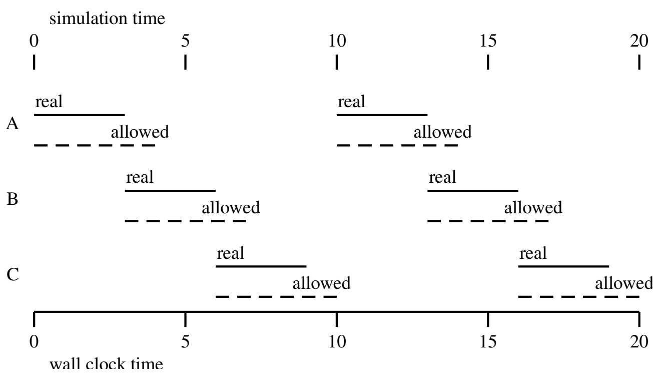


Figure K.7: Real time execution of the task chain

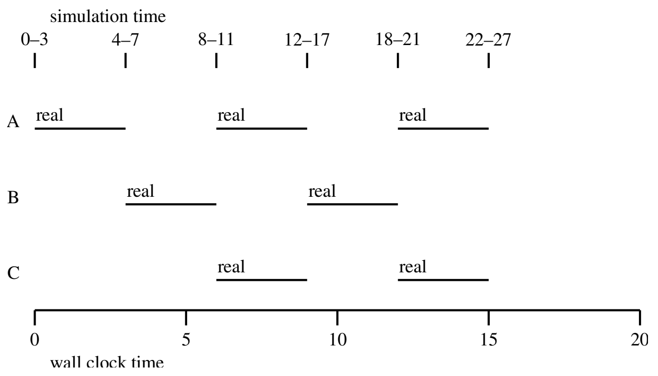


Figure K.8: AFAP execution of the task chain

After task B has completed simulation time can be incremented to 11 ms allowing task A to start again. According to the schedule this is allowed, since task C does not depend on A. The effect is that task A and C run in parallel.
If this is not the intended behavior then task C should be made dependent on task A (Figure K.9) or the sum of all allowed execution times should be made smaller then the task period.
In fact, with this schedule parallelism would also occur in the real time situation if every task had a real execution time of 4 ms.

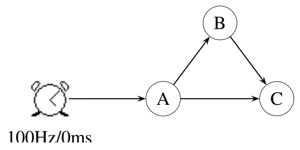


Figure K.9: A chain of dependent tasks

K.6 Other effects

Offset + allowed execution time >period

If the sum of the offset of a task and its allowed execution time is larger than the period it can happen that the task is started after a state transition.

Timed Events and Timed State Changes

In accelerated mode, Timed Events and Timed State Changes only work properly when they are expressed in simulation time. (Quite trivial.)

Non real time tasks (output connectors)

The execution delay of non real time tasks depends on the load of the system. They are not synchronized to real time tasks (by definition). It can thus happen that output connectors overflow because the accelerated periodic tasks are activating them with a too high frequency.

K.7 Performance

Estimates for the acceleration factor in AFAP scheduling can be made with the data form the timings file incremented with the scheduler overhead of Table K.1.

Activity	**Time (μs)**
Clock tick	8
Task activation	12
Empty actionMgr	17
Active Action/Recorder/Stimulus	11
Inactive Action/Recorder/Stimulus	1

Table K.1: Scheduler overhead measured on a SGI/Origin 200 R10000@225MHz with EuroSim Mk2rev2

Note that the ActionMgr has a default frequency equal to the basic frequency. This can become one of the major CPU consuming tasks in an accelerated simulation. Accelerated simulations will run faster if the ActionMgr is scheduled at a lower frequency.

K.8 Example of performance computation

	Frequency (Hz)	**Task duration (μs)**
Clock	1000	
Task A	500	100
Task B	20	500
ActionMgr	1000	
Recorder 1	100	
Recorder 2	10	

Table K.2: Example schedule on 1 CPU.

		Frequency (Hz)	**Duration (μs)**	**Subtotal (μs)**	**Total contribution (μs)**
Tasks	Task A	500	20	10000	
	Task B	20	500	10000	
	Total				10000

Table K.3: Computation time of the not optimized schedule.

		Frequency (Hz)	**Duration (μs)**	**Subtotal (μs)**	**Total contribution (μs)**
Scheduler	Clock	1000	8	8000	
	Task A	500	12	6000	
	Task B	20	12	240	
	Total				14240
ActionMgr	ActionMgr	1000	17+1+1	19000	
	Recorder 1	100	10	1000	
	Recorder 2	10	10	100	
	Total				20100
Total					44340

Table K.3: Computation time of the not optimized schedule.

Maximum acceleration of this schedule: 1000000/44340 = 23.
The actionMgr uses 20100/44340 = 45% of the computation time.
When the actionMgr is scheduled at 100 Hz it will only use 3000 μs.
The maximum acceleration will then be 1000000/27240 = 37.
This schedule could be optimized further if a basic frequency of 500 Hz is used, giving another 4000 μs reduction. The maximum acceleration will then be 1000000/23240 = 43.

Appendix L

EuroSim Mk2 to Mk3 conversion

L.1 Introduction

EuroSim Mk3 offers a completely rewritten GUI based on the Qt toolkit. A lot of effort has been invested in eliminating the shortcomings of the Mk2 interface. At the same time new features have been introduced which result in some user visible changes.
The changes for each tool will be described in a separate chapter. The last chapter describes the conversion tool which is capable of converting a complete project from Mk2 in Mk3.

L.2 Project Manager

The top-level project manager tool is completely rewritten. The old project file format has been abandoned for a more powerful database system. This means that the Mk2 database must be converted using the projconv conversion tool. This tool will convert the existing database and also the EuroSim files in each of the project directories.
The `EFO_HOME` environment variable is no longer needed. If the variable is not set, the `.eurosim` directory in the home directory of the user is used. The file name of the new project database is `projects.db` and can be found in that directory.
Each project has a file called `project.db`. This file contains a list of models, where each model has a list of associated files. This enables the user to quickly start up editors from the Project Manager tool by double clicking on the name of the file.

L.3 Model Editor

The schedule file is no longer part of the model file. The user must specify the schedule file in the new simulation definition file. Recompilation of the simulator when modifying the schedule is not needed since Mk2rev2.

L.4 Schedule Editor

The schedule editor operates now immediately on .sched files instead of the .schedule files in Mk2. This means that the conversion of .schedule files used by the editor to .sched files used by the runtime is eliminated. Mk2 .sched files are not compatible with Mk3 .sched files. Mk2 .schedule files can be converted to Mk3 .sched files with the schedule2sched conversion tool. This tool is also used when doing the conversion of a complete project. The new schedule editor is capable of loading Mk2 .schedule files.
The time bar dialog box has been removed. The overall usefulness of this dialog box is not very great. It might in fact be misleading under certain circumstances. In future releases a new analysis tool will be developed which will show precise timings of each event in the scheduler. This will much better help

user analyze timing behavior of the system. Where the Mk2 time bar display showed timings based on average, minimum and maximum values, the Mk3 variant will show individual timings.
The special EI input connector has been removed in Mk3. This is done because all external events can now be configured in the 'Tools: External Event' menu of the Schedule Editor, it was decided to not have an exception for the EI (External Interrupt) on SGI. You can simply create an external event handler for 'EI' with the appropriate path (i.e. `/dev/ei`) and set the dispatcher type to 'default'. The Schedule Editor then automatically adds an external input connector with the name of the event handler to the 'Insert: External event' menu. You can then use this input connector just as the 'EI' input connector in Mk1 and Mk2.

L.5 Mission Tool, Initial Condition Editor and Test Controller

All of these tools have been combined into the new Simulation Controller. The MDL file is no longer used as the main simulation configuration file. The role has been taken over by the simulation definition file. This file has a .sim extension. The function of this file is to combine the model file, schedule file and any number of MDL files, initial condition files and MMI (Man Machine Interface) files. Instead of the separate monitor windows in Mk2, there are now MMI panes. Each MMI pane can contain a number of alphanumerical or 2D plot monitors. These monitors can be positioned by the user on the MMI pane. Old Mk2 monitors appear on a separate MMI pane for backward compatibility. This pane is called "Script Monitors". Monitors can be copied and pasted from this pane to a new MMI pane.
It is possible to activate and deactivate individual MDL and initial condition files. If more than one initial condition file is active, this means that each initial condition is applied in the order it is listed at start-up or reset.
The Simulation Controller is capable of converting Mk2 MDL files to Mk3 Simulation Definition Files. This is described in Section 12.1.1.

L.6 Test Analyzer

The test analyzer file format has changed. The old .pdf format can be converted with the pdf2plt tool. This tool is also used when doing the conversion of a complete project. The new Test Analyzer is capable of loading Mk2 .pdf files.
The new Test Analyzer has two back-ends used to produce the plots. The user can choose to use PV-Wave or GNUplot for showing the graphs. Plotting with PV-Wave is faster than with GNUplot, but GNUplot is free.

L.7 Conversion Tool

The conversion projconv is able to convert all projects at once or individual projects. When converting all projects at once, the program is called with the Mk2 project file as an argument. The Mk2 project file is located in the directory specified in the environment variable `EFO_HOME`.
In order to convert all your EuroSim projects in one step do:

```
host:~/EfoHome$ projconv project
```

After the conversion a new project database file has been created called projects.db. If you start now the new Mk3 Project Manager you will see all the existing projects back.
It is also possible to convert individual projects. In that case you call the conversion tool like this:

```
host:~/EfoProject$ projconv .
```

After the conversion the files in that directory have been converted and a file database for this project has been created called project.db.
The conversion of a project consists of the following activities:

- build clean for every Mk2 makefile found and remove the makefile (extension .mk)
- remove Mk2 temporary model files (extension .tm)
- remove Mk2 schedule file from model file
- convert Mk2 schedule files into Mk3 sched files
- convert Mk2 plot description files (pdf) into Mk3 plot files (plt)
- create Mk3 simulation definition files from Mk2 MDL files.
- create a project file database with the following contents: model files, schedule files, simulation definition files, MDL files, initial condition files, User Program definition files.

L.8 Run-time Interface changes

The server sends some extra events compared to Mk2. The changes are as follows (see Appendix J for a full overview):

L.8.1 Real time control channel

- eventHealth returns an extra rtHealth event.

L.8.2 Mission channel

- When you join this channel you receive an extra maSimDef message with the simulation definition filename.
- eventNewMission returns an extra maNewMission event.
- eventOpenMission returns an extra maOpenMission event.
- eventNewAction returns an extra maNewAction event.
- eventDeleteAction returns an extra maDeleteAction event.
- eventReload returns an extra maReload event.

L.8.3 Monitor channel

- eventAdd2LogList returns an extra dtAdd2LogList event.
- eventRemoveFromLogList returns an extra dtRemoveFromLogList event.
- eventSetValueRequest returns an extra dtSetValueRequest event.
- eventCpuLoadSetPeak returns an extra dtCpuLoadSetPeak event.

L.8.4 Scheduler control channel

- eventWhere now always returns the current position in the schedule (this command was undocumented in Mk2 and in fact didn't always return the position).
- eventRaiseEvent returns an extra scRaiseEvent event.
- eventRaiseEventAt returns an extra scRaiseEventAt event.
- eventRaiseEventAtSimtime returns an extra scRaiseEventAtSimtime event.

Appendix M

Introduction to CVS

M.1 Introduction

CVS, short for Concurrent Versions System, allows you to save versions of your files for later retrieval by yourself or other users (provided they have sufficient access rights). The files are stored in what is called a "repository". This chapter describes the basic commands that are required to start using CVS with EuroSim. See [CVS00] for more information on CVS.

M.2 Initializing the repository root

After deciding where to install the CVS repository root (usually a directory on a network drive that is backed-up at regular intervals), you must initialize it:

- Open a shell and change directory to the designated directory (create it first if it doesn't exist yet):

  `cd` *repository_root_directory*

- Set the `CVSROOT` environment variable:

  `export CVSROOT=`*repository_root_directory*

  Example for IRIX/Linux:

  ```
  export CVSROOT=/projects/share/repository
  ```

  See Section M.4 for a description on how to use CVS under Windows.

- Initialize the CVS repository:

  ```
  cvs init
  ```

If all went well, a `CVSROOT` directory is created in the *repository_root_directory*. Note that you only have to perform the above steps once.

M.3 Setting up a CVS repository

Once the CVS repository root has been initialized, you can add "repositories" to it. When using CVS with EuroSim, you can create a repository for the directory where your project files are located:

- Go to the directory where the files of your EuroSim project are located (model files, schedule file, etc...).

  `cd` *project_directory*

- Create an empty CVS repository directory in the CVS repository root:

  `cvs import -I \* -m` *log_msg repository vendor_tag release_tag*

  The `-I` option with the escaped wildcard (`\*`) tells CVS to ignore all files in the project directory. This is done because at this point we do not want to import any files into the repository: we selectively add files to the repository later on by means of the menu commands in the EuroSim tools.

  The `-m` option allows you to enter a descriptive log message for the repository. Enclose the message in quotes or double quotes.

  The *vendor_tag* and *release_tag* can be any text, because we are not importing any files at this point.

  Example:

  ```
  cvs import -I \* -m 'Test' MyProject Foo Bar
  ```

- Go to the parent directory

  ```
  cd ..
  ```

- Initialize *project_directory* with the CVS files:

  `cvs checkout -d` *project_directory repository_name*

  The project directory should now contain a directory `CVS`.

  Example:

  ```
  cvs checkout -d MyProject MyProject
  ```

You can now start the EuroSim Project Manager, select your project and select the *Tools:Project Settings* menu command to set the project repository root to the *repository_root_directory* that you assigned to the `CVSROOT` environment variable. When starting the EuroSim tools from the Project Manager, you can use the *Tools:Version* menu commands to add files to the repository.

M.4 Using CVS under Windows

When you are using Cygwin's native version of CVS, then specify the `CVSROOT` environment variable as follows:

`export CVSROOT=/cygdrive/`*drive_letter*`/`*repository_root_directory*

Example for Cygwin when your repository is on the `F:` drive

```
export CVSROOT=/cygdrive/f/repository
```

Other versions of CVS for Windows may require the addition of the *local* server specification like this:

`export CVSROOT=:local:`*drive_letter*`/`*repository_root_directory*

For example:

```
export CVSROOT=:local:F:/repository
```

Consult the README files of the version of CVS that you are using for more information on how to set up CVS.

M.5 More information

You can get more information by typing:

```
man cvs
```

on the command line. Of course the internet provides multiple sources of CVS manuals in multiple formats (.tex, .pdf, etc. . .). O'Reilly & Associates have a nice pocket reference, see [CVS00].

Appendix N

EuroSim XML Schemas

The XML files used in EuroSim are officially described by XML schemas. These schema files are located in the `lib/schemas` subdirectory of the EuroSim installation directory.

Appendix O

Software Problem Reports

In case a problem or error with EuroSim occurs, use the `spr` tool for submitting Software Problem Reports. See Figure O.1 for a screendump of the tool.
Document the problem or error with as much detail as possible. Things of interest are:

- Software version numbers
- Hardware specifications
- Sequence of actions (such as selecting files, clicking buttons, changing state of the simulator)
- Contents of files used

Preferably the problem or error should be reproducible, and try to create a minimal environment in which the error occurs, to facilitate finding the source of the problem.
The user criticality can be one of:

Critical A major problem that hinders the completion of the user's job. This category includes a time aspect (solution is needed as soon as possible) for the user to be able to finish the job.

Major A serious problem, but the user can still continue with the job.

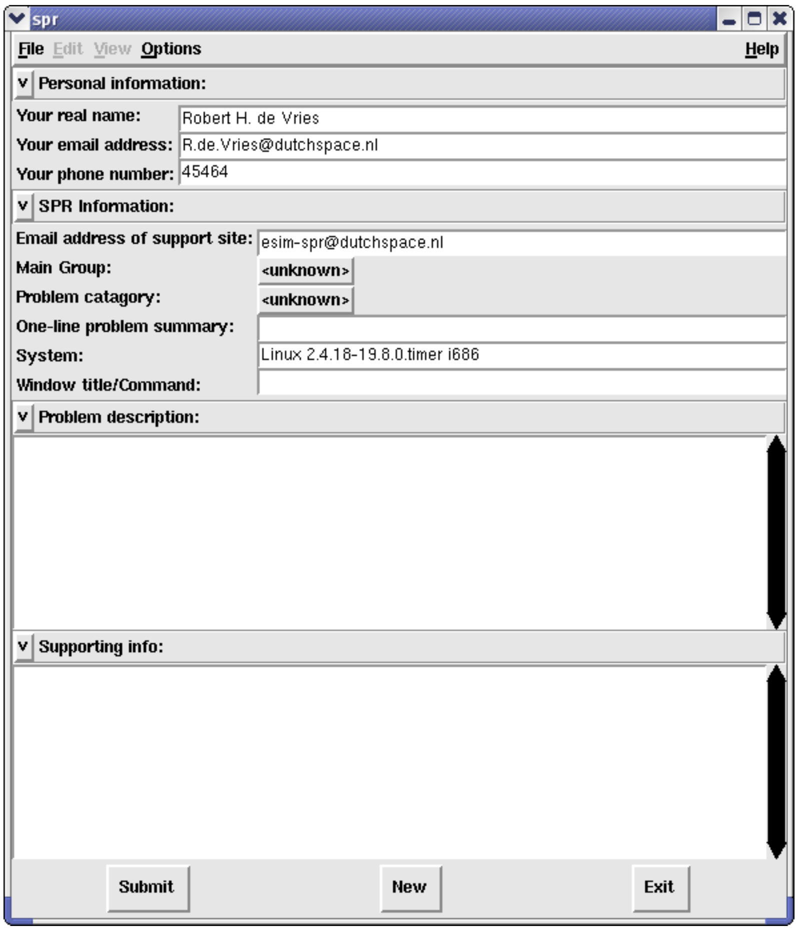
Minor A problem was noted, but it is not seriously affecting the use of EuroSim.

Suggestion
A suggestion for the improvement of EuroSim.

Question
A question on EuroSim details.

If you are not able to submit the SPR by e-mail, then please send a paper version and any related information to:

EuroSim Product Support
Dutch Space BV
P.O. Box 32070
2303 DB Leiden
The Netherlands



Personal information:

Your real name:	Robert H. de Vries
Your email address:	R.de.Vries@dutchspace.nl
Your phone number:	45464

SPR Information:

Email address of support site:	esim-spr@dutchspace.nl
Main Group:	<unknown>
Problem catagory:	<unknown>
One-line problem summary:	
System:	Linux 2.4.18-19.8.0.timer i686
Window title/Command:	

Problem description:

Supporting info:

Figure O.1: The SPR tool

Bibliography

[COM98] *Inside distributed COM*, 1998, ISBN 1-57231-849-X, Microsoft Press, Eddon & Eddon. Background on (D)COM components and applications.

[CVS00] *CVS pocket reference*, 2000, ISBN 0-596-00003-0, O'Reilly & Associates, Gregor N. Purdy. Pocket reference to the Concurrent Versions System.

[FAQ05] *EuroSim frequently asked questions*, 2005, This can be found in `$EFOROOT/doc/html/FAQ/faq.html`. This file contains the EuroSim Frequently Asked Questions list in HTML format.

[MAN05] *EuroSim manual pages*, 2005, Stored in `$EFOROOT/man`. This directory contains the EuroSim on-line manual pages, which can be read using the UNIX `man` command.

[OM05] Dutch Space BV, *EuroSim Mk4.0 owner's manual*, 2005, FSS-EFO-TN-530. This document contains the information relevant for the facility manager of EuroSim. Stored in `$EFOROOT/doc/pdf/OM.pdf`. This file contains the EuroSim Owner's Manual in Adobe Acrobat format. Also stored in directory `$EFOROOT/doc/html/OM`. This directory contains the EuroSim Owner's Manual in HTML format.

[PMA05] *EuroSim manual pages*, 2005, FSS-EFO-SPE-523, issue 3 revision 0, 2-Sep-2004. This document contains a printed version of all end user relevant manual pages, which are also available on-line though the UNIX `man` command.

[PVW] Visual Numerics, Inc., *Documentation and manuals for PV-WAVE CL version 6.01*, Contains the user manual and reference documentation for the operation of PV-Wave.

[Sec03] ECSS Secretariat (ed.), *Ground systems and operations - telemetry and telecommand packet utilization*, Space engineering, no. ECSS-E-70-41A, ESA-ESTEC, 2003.

[SMP03] *Simulation model portability handbook*, 2003, EWP-2080, issue 1, revision 4, 2003/01/29. This document is the Handbook for the Simulation Model Portability (SMP) Standard.

[SMP05a] *Simulation model portability 2.0 c++ mapping*, 2005, EGOS-SIM-GEN-TN-0102, issue 1, revision 2, 2005/10/28. This document contains the mapping to C++ for both the metamodel and the component model of the SMP2 standard.

[SMP05b] *Simulation model portability 2.0 component model*, 2005, EGOS-SIM-GEN-TN-0101, issue 1, revision 2, 2005/10/28. This document specifies the component model of the SMP2 standard.

[SMP05c] *Simulation model portability 2.0 handbook*, 2005, EGOS-SIM-GEN-TN-0099, issue 1, revision 2, 2005/10/28. This document is the Handbook for the SMP2 Standard.

[SMP05d] *Simulation model portability 2.0 c++ model development kit*, 2005, EGOS-SIM-GEN-TN-1001, issue 1, revision 2, 2005/10/28. This document contains the documentation of the Model Development Kit for the SMP2 standard.

[SMP05e] *Simulation model portability 2.0 metamodel*, 2005, EGOS-SIM-GEN-TN-0100, issue 1, revision 2, 2005/10/28. This document describes the metamodel specification (SMDL) of the SMP2 standard.

[SPR05] *Resolved* SPR *list*, 2005, Stored in `$EFOROOT/etc/ResolvedSPRList`. This file contains a list of solved bugs (SPRs) of each EuroSim release.

[SRN05] *EuroSim Mk4.0 software release notes*, 2005, FSS-EFO-SRN-388. Stored in `$EFOROOT/etc/SoftwareReleaseNote`. Final word from developers before packaging; always contains last and latest information concerning delivered EuroSim release.

[SUM05] Dutch Space BV, *EuroSim Mk4.0 software user's manual*, 2005, NLR-EFO-SUM-002. Stored in `$EFOROOT/doc/pdf/SUM.pdf`. This file contains the EuroSim Software User Manual in Adobe Acrobat format. Also stored in directory `$EFOROOT/doc/html/SUM`. This directory contains the EuroSim Software User Manual in HTML format.

[VMI] *VMIVME-6000 BCU software library*.

[VMI93] *VMIVME-6000, 1553 communications interface board, product manual*, October 26 1993, These documents contain information on the VMIVME-6000 BCU software library.

Index